

Type Annotations with Roundtrip Property

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

June 25, 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

1	Human-authored proof formalization	2
1.1	Types	2
1.1.1	Type variables of a type	2
1.1.2	Type substitution	3
1.1.3	Substitution composition and single substitution	3
1.1.4	Basic properties of substitution	3
1.2	Terms	4
1.2.1	Type variables of a term	4
1.2.2	Substitution on terms	5
1.2.3	Term predicates: F-term, U-term, C-term	6
1.2.4	Unambiguity	7
1.3	Positions	8
1.3.1	Removing annotations	9
1.4	Completion relations for types and terms	10

1.4.1	Inversion rules	10
1.4.2	Basic properties	11
1.4.3	Positions and completions	12
1.4.4	Well-typedness and type inference	20
1.4.5	Well-Typed Completions	23
1.5	Correct Printings	24
1.6	The Algorithm	27
1.7	Completeness	29
1.8	Minimality	33
2	AI-Authored proof formalization of the roundtrip algorithm	36
2.1	Types and Type Substitutions	36
2.2	Contexts	37
2.3	Terms	38
2.3.1	Raw Terms	38
2.3.2	Annotated Terms	39
2.3.3	Well-Typedness	40
2.4	Positions and Post-Order Enumeration	41
2.4.1	Basic Properties of Enumeration	42
2.4.2	Substitution Distributes to Positions	43
2.4.3	Distinctness and Range of Position Numbers	44
2.5	The Annotation Algorithm	45
2.5.1	Type Inference Assumption	45
2.5.2	Annotations determine the substitution	47
2.5.3	Completeness	48
2.5.4	Local Minimality	49
2.6	Annotation Insertion	51
2.7	The Reverse-Greedy Algorithm	52
2.7.1	Basic Properties of the Fold	53
2.7.2	Coverage	53
2.7.3	Witness Property	57
2.7.4	Connecting the Algorithm to the Locale	61
3	AI-Authored proof formalization via independence systems	66
3.1	Independence Systems	66
3.2	Best-In-Greedy Algorithm	66
3.2.1	Properties of <i>greedy-partial</i>	67
3.2.2	Theorem: Best-In-Greedy returns a maximal independent set	68
3.3	The Annotation Set System	70
3.3.1	Setup	70
3.3.2	The Independence System	70
3.3.3	The annotation set system is an independence system	71
3.4	Feasibility and Count Condition	71

3.5	Local Minimality	73
3.6	Reverse Greedy Correctness	73

```

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: $\sigma ::= \alpha | \sigma \Rightarrow \tau | (\sigma_1, \dots, \sigma_n) \kappa$ 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: $(\text{'tv}, \text{'k}) \text{ ty option}$ where *None* represents \perp (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 tvs-ty :: ('tv, 'k) ty  $\Rightarrow$  'tv set where
  tvs-ty (TyVar a) = {a}
| tvs-ty (Arrow  $\sigma$   $\tau$ ) = tvs-ty  $\sigma$   $\cup$  tvs-ty  $\tau$ 
| tvs-ty (TyCon  $\kappa$   $\sigma$ s) =  $\bigcup$  (tvs-ty ' set  $\sigma$ s)

```

```

definition tvs-mty :: ('tv, 'k) mty  $\Rightarrow$  'tv set where
  tvs-mty  $\xi$  = (case  $\xi$  of None  $\Rightarrow$   $\{\}$  | Some  $\sigma$   $\Rightarrow$  tvs-ty  $\sigma$ )

```

```

lemma tvs-mty-None[simp]: tvs-mty None =  $\{\}$ 
by (simp add: tvs-mty-def)

```

```

lemma tvs-mty-Some[simp]: tvs-mty (Some  $\sigma$ ) = tvs-ty  $\sigma$ 
by (simp add: tvs-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 $\rho \sigma$* applies ρ to σ . Extended to maybe-types with *subst-mty $\rho None = None$* .

fun *subst-ty* :: (*'tv \Rightarrow ('tv, 'k) ty*) \Rightarrow (*'tv, 'k*) *ty* \Rightarrow (*'tv, 'k*) *ty* **where**
subst-ty ρ (TyVar a) = ρ a
| *subst-ty ρ (Arrow $\sigma \tau$) = Arrow (subst-ty $\rho \sigma$) (subst-ty $\rho \tau$)*
| *subst-ty ρ (TyCon $\kappa \sigma s$) = TyCon κ (map (subst-ty ρ) σs)*

definition *subst-mty* :: (*'tv \Rightarrow ('tv, 'k) ty*) \Rightarrow (*'tv, 'k*) *mty* \Rightarrow (*'tv, 'k*) *mty* **where**
subst-mty $\rho \xi = map-option (subst-ty \rho) \xi$

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

lemma *subst-mty-Some*[simp]: *subst-mty $\rho (Some \sigma) = Some (subst-ty \rho \sigma)$*
by (*simp add: subst-mty-def*)

1.1.3 Substitution composition and single substitution

definition *comp-subst* ::
(*'tv \Rightarrow ('tv, 'k) ty*) \Rightarrow (*'tv \Rightarrow ('tv, 'k) ty*) \Rightarrow (*'tv \Rightarrow ('tv, 'k) ty*) (**infixl** $\circ\circ$ 55)
where
(*$\rho \circ\circ \rho'$*) *a = subst-ty $\rho (\rho' a)$*

Composition $\rho \circ\circ \rho'$ is defined by $(\rho \circ\circ \rho') a = \text{subst-ty } \rho (\rho' a)$. Single substitution *single-subst $\tau \alpha$* maps α to τ and everything else to itself.

definition *single-subst* :: (*'tv, 'k*) *ty* \Rightarrow *'tv \Rightarrow ('tv, 'k) ty* **where**
single-subst $\tau \alpha = (\lambda a. \text{if } a = \alpha \text{ then } \tau \text{ else TyVar } a)$

lemma *single-subst-same*[simp]: *single-subst $\tau \alpha \alpha = \tau$*
by (*simp add: single-subst-def*)

lemma *single-subst-other*[simp]: *a \neq $\alpha \implies \text{single-subst } \tau \alpha a = \text{TyVar } a$*
by (*simp add: single-subst-def*)

1.1.4 Basic properties of substitution

lemma *subst-ty-comp*:
subst-ty $\rho (\text{subst-ty } \rho' \tau) = \text{subst-ty } (\rho \circ\circ \rho') \tau$
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* ($\rho \circ \rho'$) ξ
by (*cases* ξ) (*auto simp: subst-mty-def subst-ty-comp comp-subst-def*)

lemma *subst-ty-agree*:
subst-ty ρ $\tau = \text{subst-ty } \rho' \tau \iff (\forall \alpha \in \text{tvars-ty } \tau. \rho \alpha = \rho' \alpha)$
by (*induction* τ) *auto*

lemma *subst-ty-cong*:
 $(\bigwedge \alpha. \alpha \in \text{tvars-ty } \tau \implies \rho \alpha = \rho' \alpha) \implies \text{subst-ty } \rho \tau = \text{subst-ty } \rho' \tau$
using *subst-ty-agree* **by** *blast*

lemma *subst-ty-swap*:
assumes $\beta \notin \text{tvars-ty } \tau$
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) = $\bigcup (\text{tvars-ty } \rho \text{ ` tvars-ty } \tau)$
by (*induction* τ) *auto*

lemma *tvars-mty-subst*:
tvars-mty (*subst-mty* ρ ξ) = $\bigcup (\text{tvars-ty } \rho \text{ ` tvars-mty } \xi)$
by (*cases* ξ) (*auto simp: tvars-ty-subst*)

1.2 Terms

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

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

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* \Rightarrow *'tv* *set* **where**
tvars-trm (*Vr* $x \ \xi$) = *tvars-mty* ξ
| *tvars-trm* (*Ct* $c \ \xi$) = *tvars-mty* ξ

| $tvars\text{-}trm (App\ t1\ t2\ \xi) = tvars\text{-}trm\ t1 \cup tvars\text{-}trm\ t2 \cup tvars\text{-}mtty\ \xi$
| $tvars\text{-}trm (Lam\ x\ \xi\ t\ \zeta) = tvars\text{-}mtty\ \xi \cup tvars\text{-}trm\ t \cup tvars\text{-}mtty\ \zeta$

lemma $tvars\text{-}trm\text{-}finite[simp, intro]: finite (tvars\text{-}trm\ t)$
by ($induction\ t$) $auto$

1.2.2 Substitution on terms

$subst\text{-}trm\ \rho\ t$: apply type substitution ρ to all type annotations in t .

fun $subst\text{-}trm :: ('tv \Rightarrow ('tv, 'k)\ ty) \Rightarrow ('tv, 'k, 'v, 'c)\ trm \Rightarrow ('tv, 'k, 'v, 'c)\ trm$ **where**
 $subst\text{-}trm\ \rho (Vr\ x\ \xi) = Vr\ x (subst\text{-}mtty\ \rho\ \xi)$
| $subst\text{-}trm\ \rho (Ct\ c\ \xi) = Ct\ c (subst\text{-}mtty\ \rho\ \xi)$
| $subst\text{-}trm\ \rho (App\ t1\ t2\ \xi) = App (subst\text{-}trm\ \rho\ t1) (subst\text{-}trm\ \rho\ t2) (subst\text{-}mtty\ \rho\ \xi)$
| $subst\text{-}trm\ \rho (Lam\ x\ \xi\ t\ \zeta) = Lam\ x (subst\text{-}mtty\ \rho\ \xi) (subst\text{-}trm\ \rho\ t) (subst\text{-}mtty\ \rho\ \zeta)$

lemma $subst\text{-}trm\text{-}comp$:
 $subst\text{-}trm\ \rho (subst\text{-}trm\ \rho'\ t) = subst\text{-}trm (\rho \circ \rho') t$
by ($induction\ t$) ($auto\ simp: subst\text{-}mtty\text{-}comp$)

lemma $subst\text{-}trm\text{-}id[simp]: subst\text{-}trm\ TyVar\ t = t$
by ($induction\ t$) $auto$

lemma $subst\text{-}trm\text{-}cong$:
 $(\bigwedge \alpha. \alpha \in tvars\text{-}trm\ t \implies \rho\ \alpha = \rho'\ \alpha) \implies subst\text{-}trm\ \rho\ t = subst\text{-}trm\ \rho'\ t$
proof ($induction\ t$)
case ($Vr\ x\ \xi$)
then show $?case$ **by** ($cases\ \xi$) ($auto\ intro: subst\text{-}ty\text{-}cong$)
next
case ($Ct\ c\ \xi$)
then show $?case$ **by** ($cases\ \xi$) ($auto\ intro: subst\text{-}ty\text{-}cong$)
next
case ($App\ t1\ t2\ \xi$)
then show $?case$ **by** ($cases\ \xi$) ($auto\ cong: subst\text{-}ty\text{-}cong$)
next
case ($Lam\ x\ \xi\ t\ \zeta$)
then show $?case$ **by** ($cases\ \xi; cases\ \zeta$) ($auto\ intro: subst\text{-}ty\text{-}cong$)
qed

lemma $subst\text{-}trm\text{-}agree$:
 $subst\text{-}trm\ \rho\ t = subst\text{-}trm\ \rho'\ t \iff (\forall \alpha \in tvars\text{-}trm\ t. \rho\ \alpha = \rho'\ \alpha)$
proof ($induction\ t$)
case ($Vr\ x\ \xi$)
then show $?case$ **by** ($cases\ \xi$) ($auto\ simp: subst\text{-}ty\text{-}agree$)
next
case ($Ct\ c\ \xi$)
then show $?case$ **by** ($cases\ \xi$) ($auto\ simp: subst\text{-}ty\text{-}agree$)
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 $\text{subst-trm } \rho v = v$, then ρ acts as identity on $\text{tvars-trm } v$.

lemma *subst-trm-id-on-tvars*:

```

  assumes subst-trm ρ v = v α ∈ 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 ∀α ∈ 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 β ∉ 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) = ⋃(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) ⟷ ξ ≠ None

```

$| \text{fterm } (Ct\ c\ \xi) \longleftrightarrow \xi \neq \text{None}$
 $| \text{fterm } (App\ t1\ t2\ \xi) \longleftrightarrow \text{fterm } t1 \wedge \text{fterm } t2 \wedge \xi \neq \text{None}$
 $| \text{fterm } (Lam\ x\ \xi\ t\ \zeta) \longleftrightarrow \xi \neq \text{None} \wedge \text{fterm } t \wedge \zeta \neq \text{None}$

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

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

lemma *fterm-subst[simp]*: $\text{fterm } v \Longrightarrow \text{fterm } (\text{subst-trm } \varrho\ v)$
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**
 $\text{bvars } (Vr\ x\ \xi) = \{\}$
 $| \text{bvars } (Ct\ c\ \xi) = \{\}$
 $| \text{bvars } (App\ t1\ t2\ \xi) = \text{bvars } t1 \cup \text{bvars } t2$
 $| \text{bvars } (Lam\ x\ \xi\ t\ \zeta) = \{x\} \cup \text{bvars } t$

lemma *subst-trm-bvars[simp]*: $\text{bvars } (\text{subst-trm } \varrho\ t) = \text{bvars } t$
by (*induction t*) *auto*

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

lemma *subst-trm-unambiguous[simp]*: $\text{unambiguous } (\text{subst-trm } \varrho\ t) = \text{unambiguous } t$
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. $mtp\text{-of}\ t\ p$: the maybe-type at position p in t .

fun $poss :: ('tv, 'k, 'v, 'c)\ trm \Rightarrow nat\ list\ set$ **where**
 $poss\ (Vr\ x\ \xi) = \{\}\}$
 $|\ poss\ (Ct\ c\ \xi) = \{\}\}$
 $| \ poss\ (App\ t1\ t2\ \xi) = \{\}\} \cup ((\lambda p. 1 \# p) \text{ ' } poss\ t1) \cup ((\lambda p. 2 \# p) \text{ ' } poss\ t2)$
 $| \ poss\ (Lam\ x\ \xi\ t\ \zeta) = \{\}\} \cup \{[1]\} \cup ((\lambda p. 2 \# p) \text{ ' } poss\ t)$

lemma $poss\text{-finite}[simp, intro]: finite\ (poss\ t)$
by $(induction\ t)\ auto$

lemma $nil\text{-in}\text{-}poss[simp]: [] \in poss\ t$
by $(cases\ t)\ auto$

lemma $subst\text{-}trm\text{-}poss[simp]: poss\ (subst\text{-}trm\ \varrho\ t) = poss\ t$
by $(induction\ t)\ auto$

fun $mtp\text{-of} :: ('tv, 'k, 'v, 'c)\ trm \Rightarrow nat\ list \Rightarrow ('tv, 'k)\ mty$ **where**
 $mtp\text{-of}\ (Vr\ x\ \xi)\ p = \xi$
 $| \ mtp\text{-of}\ (Ct\ c\ \xi)\ p = \xi$
 $| \ mtp\text{-of}\ (App\ t1\ t2\ \xi)\ p = (case\ p\ of$
 $\ \ [] \Rightarrow \xi$
 $\ \ | \ (Suc\ 0\ \# \ p') \Rightarrow mtp\text{-of}\ t1\ p'$
 $\ \ | \ (Suc\ (Suc\ 0)\ \# \ p') \Rightarrow mtp\text{-of}\ t2\ p'$
 $\ \ | \ - \Rightarrow None)$
 $| \ mtp\text{-of}\ (Lam\ x\ \xi\ t\ \zeta)\ p = (case\ p\ of$
 $\ \ [] \Rightarrow \zeta$
 $\ \ | \ [Suc\ 0] \Rightarrow \xi$
 $\ \ | \ (Suc\ (Suc\ 0)\ \# \ p') \Rightarrow mtp\text{-of}\ t\ p'$
 $\ \ | \ - \Rightarrow None)$

$mtpOf\ t$ is the maybe-type of the root of t .

abbreviation $mtp\text{-of}\text{-}root :: ('tv, 'k, 'v, 'c)\ trm \Rightarrow ('tv, 'k)\ mty$ **where**
 $mtp\text{-of}\text{-}root\ t \equiv mtp\text{-of}\ t\ []$

lemma $mtp\text{-of}\text{-}subst$:
assumes $p \in poss\ t$
shows $mtp\text{-of}\ (subst\text{-}trm\ \varrho\ t)\ p = subst\text{-}mty\ \varrho\ (mtp\text{-of}\ t\ p)$
using $assms$ **by** $(induction\ t\ arbitrary: p)\ (auto\ split: list.splits\ nat.splits)$

lemma $tvars\text{-}trm\text{-}eq\text{-}UN\text{-}mtp$:
 $tvars\text{-}trm\ t = (\bigcup p \in poss\ t. tvs\text{-}mty\ (mtp\text{-of}\ t\ p))$
by $(induction\ t)\ (auto\ simp: image\text{-}iff)$

lemma $fterm\text{-}mtp\text{-of}\text{-}Some$:
assumes $fterm\ v\ p \in poss\ v$
shows $\exists \sigma. mtp\text{-of}\ v\ p = Some\ \sigma$

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

| $erase (Lam\ x\ \xi\ t\ \zeta) = Lam\ x\ None\ (erase\ t)\ None$

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

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

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

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

lemma $mcompl\ antisym$: $\llbracket \xi \sqsubseteq_m \zeta; \zeta \sqsubseteq_m \xi \rrbracket \implies \xi = \zeta$
by $(auto\ simp:\ mcompl\ def)$

lemma $mcompl\ trans$: $\llbracket \xi \sqsubseteq_m \zeta; \zeta \sqsubseteq_m \chi \rrbracket \implies \xi \sqsubseteq_m \chi$
by $(auto\ simp:\ mcompl\ def)$

inductive $compl :: ('tv, 'k, 'v, 'c)\ trm \Rightarrow ('tv, 'k, 'v, 'c)\ trm \Rightarrow bool$ (**infix** \sqsubseteq 50)
where

$compl\ Vr$: $\xi \sqsubseteq_m \zeta \implies Vr\ x\ \xi \sqsubseteq Vr\ x\ \zeta$
| $compl\ Ct$: $\xi \sqsubseteq_m \zeta \implies Ct\ c\ \xi \sqsubseteq Ct\ c\ \zeta$
| $compl\ App$: $\llbracket s1 \sqsubseteq s1'; s2 \sqsubseteq s2'; \xi \sqsubseteq_m \xi' \rrbracket \implies App\ s1\ s2\ \xi \sqsubseteq App\ s1'\ s2'\ \xi'$
| $compl\ Lam$: $\llbracket \zeta \sqsubseteq_m \zeta'; t \sqsubseteq t'; \xi \sqsubseteq_m \xi' \rrbracket \implies Lam\ x\ \zeta\ t\ \xi \sqsubseteq Lam\ x\ \zeta'\ t'\ \xi'$

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** $\xi \sqsubseteq_m \xi' \wedge t = Vr\ x\ \xi'$
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*: $t \sqsubseteq s \implies poss\ t = poss\ s$

by (*induction t s rule: compl.induct*) *auto*

lemma *compl-bvars-eq*: $t \sqsubseteq s \implies bvars\ t = bvars\ s$

by (*induction t s rule: compl.induct*) *auto*

lemma *compl-unambiguous*: $t \sqsubseteq s \implies unambiguous\ t = unambiguous\ s$

by (*induction t s rule: compl.induct*) (*auto simp: compl-bvars-eq*)

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

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

lemma *compl-trans*: $\llbracket s \sqsubseteq t; t \sqsubseteq u \rrbracket \implies s \sqsubseteq u$
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*: $\llbracket s \sqsubseteq t; t \sqsubseteq s \rrbracket \implies s = t$
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*: $t \sqsubseteq s \implies \text{mtp-of } t \sqsubseteq_m \text{mtp-of } s$
by (*erule compl.cases*) *auto*

1.4.3 Positions and completions

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

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

lemma *erase-compl-remove-annot*:
assumes $\text{erase } t \sqsubseteq s$
shows $\text{erase } t \sqsubseteq \text{remove-annot } s$
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.prem*(1) **obtain** $s1 \ s2 \ \xi s$ **where** $\text{seq: } s = \text{App } s1 \ s2 \ \xi s$
 $\text{erase } t1 \sqsubseteq s1 \ \text{erase } t2 \sqsubseteq s2$
by (*auto elim!: compl-App-leftE*)
show *?case* **using** *App.prem seq*
by (*cases p*) (*auto split: nat.splits intro: compl.intros App.IH*)

```

next
  case (Lam x  $\xi$  t  $\zeta$ )
  from Lam.prems(1) obtain s0  $\zeta$  s  $\xi$  s where seq: s = Lam x  $\zeta$  s s0  $\xi$  s
    erase t  $\sqsubseteq$  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 $t \sqsubseteq u$ $t \sqsubseteq u'$ *fterm* *u* *fterm* *u'* $\wedge p. p \in \text{poss } t \implies \text{mtp-of } u \text{ } p = \text{mtp-of } u' \text{ } p$

shows $u = u'$

using *assms*

proof (*induction* *t* *arbitrary*: $u \ u'$)

case (*Vr* *x* ξ)

from *Vr.prem*s(1) **obtain** ξ *u* **where** $u = \text{Vr } x \ \xi \ u$ **by** (*auto elim!*: *compl-Vr-leftE*)
moreover from *Vr.prem*s(2) **obtain** ξ *u'* **where** $u' = \text{Vr } x \ \xi \ u'$ **by** (*auto elim!*: *compl-Vr-leftE*)

ultimately show $u = u'$ **using** *Vr.prem*s(5)[of []] **by** *auto*

next

case (*Ct* *c* ξ)

from *Ct.prem*s(1) **obtain** ξ *u* **where** $u = \text{Ct } c \ \xi \ u$ **by** (*auto elim!*: *compl-Ct-leftE*)
moreover from *Ct.prem*s(2) **obtain** ξ *u'* **where** $u' = \text{Ct } c \ \xi \ u'$ **by** (*auto elim!*: *compl-Ct-leftE*)

ultimately show $u = u'$ **using** *Ct.prem*s(5)[of []] **by** *auto*

next

case (*App* *t1* *t2* ξ)

from *App.prem*s(1) **obtain** *u1* *u2* ξ *u* **where** *ueq*: $u = \text{App } u1 \ u2 \ \xi \ u$
 $t1 \sqsubseteq u1$ $t2 \sqsubseteq u2$ **by** (*auto elim!*: *compl-App-leftE*)

from *App.prem*s(2) **obtain** *u1'* *u2'* ξ *u'* **where** *u'eq*: $u' = \text{App } u1' \ u2' \ \xi \ u'$
 $t1 \sqsubseteq u1'$ $t2 \sqsubseteq u2'$ **by** (*auto elim!*: *compl-App-leftE*)

have *fu*: *fterm* *u1* *fterm* *u2* **using** *App.prem*s(3) *ueq*(1) **by** *auto*

have *fu'*: *fterm* *u1'* *fterm* *u2'* **using** *App.prem*s(4) *u'eq*(1) **by** *auto*

have *eq1*: $\wedge p. p \in \text{poss } t1 \implies \text{mtp-of } u1 \text{ } p = \text{mtp-of } u1' \text{ } p$

proof –

fix *p* **assume** $p \in \text{poss } t1$

then have *Suc* 0 # $p \in \text{poss } (\text{App } t1 \ t2 \ \xi)$ **by** *auto*

from *App.prem*s(5)[*OF this*] *ueq*(1) *u'eq*(1) **show** $\text{mtp-of } u1 \text{ } p = \text{mtp-of } u1' \text{ } p$

by *auto*

qed

have *eq2*: $\wedge p. p \in \text{poss } t2 \implies \text{mtp-of } u2 \text{ } p = \text{mtp-of } u2' \text{ } p$

proof –

fix *p* **assume** $p \in \text{poss } t2$

then have 2 # $p \in \text{poss } (\text{App } t1 \ t2 \ \xi)$ **by** *auto*

from *App.prem*s(5)[*OF this*] *ueq*(1) *u'eq*(1) **show** $\text{mtp-of } u2 \text{ } p = \text{mtp-of } u2' \text{ } p$

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 $\xi u = \xi u'$ **using** $App.prem(5)[of\ []]\ ueq(1)\ u'eq(1)$ **by** *auto*
ultimately show $u = u'$ **using** $ueq(1)\ u'eq(1)$ **by** *auto*

next
case $(Lam\ x\ \xi\ t\ \zeta)$
from $Lam.prem(1)$ **obtain** $u0\ \zeta u\ \xi u$ **where** $ueq: u = Lam\ x\ \zeta u\ u0\ \xi u$
 $t \sqsubseteq u0$ **by** $(auto\ elim!: compl-Lam-leftE)$
from $Lam.prem(2)$ **obtain** $u0'\ \zeta u'\ \xi u'$ **where** $u'eq: u' = Lam\ x\ \zeta u'\ u0'\ \xi u'$
 $t \sqsubseteq u0'$ **by** $(auto\ elim!: compl-Lam-leftE)$
have $fu: fterm\ u0$ **using** $Lam.prem(3)\ ueq(1)$ **by** *auto*
have $fu': fterm\ u0'$ **using** $Lam.prem(4)\ u'eq(1)$ **by** *auto*
have $eq0: \bigwedge p. p \in poss\ t \implies mtp-of\ u0\ p = mtp-of\ u0'\ p$
proof –
fix p **assume** $p \in poss\ t$
then have $2 \# p \in poss\ (Lam\ x\ \xi\ t\ \zeta)$ **by** *auto*
from $Lam.prem(5)[OF\ this]\ ueq(1)\ u'eq(1)$ **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 $\zeta u = \zeta u'$ **using** $Lam.prem(5)[of\ [Suc\ 0]]\ ueq(1)\ u'eq(1)$ **by** *auto*
moreover have $\xi u = \xi u'$ **using** $Lam.prem(5)[of\ []]\ ueq(1)\ u'eq(1)$ **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 $compl-u: t \sqsubseteq u$ **and** $fu: fterm\ u$
and $inst-u: \exists \rho. u = subst-trm\ \rho\ v$ **and** $fv: fterm\ v$
and $cover: \forall \alpha \in tvars-trm\ v. \exists p \in poss\ v. \alpha \in tvars-mty\ (mtp-of\ v\ p) \wedge mtp-of\ t\ p \neq None$
and $compl-u': t \sqsubseteq u'$ **and** $fu': fterm\ u'$
and $inst-u': \exists \rho'. u' = subst-trm\ \rho'\ v$
shows $u = u'$

proof –

from $inst-u$ **obtain** ρ **where** $u-eq: u = subst-trm\ \rho\ v$ **by** *auto*
from $inst-u'$ **obtain** ρ' **where** $u'-eq: u' = subst-trm\ \rho'\ v$ **by** *auto*

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

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** $p\text{-in}: p \in \text{poss } t$
then have $pv: p \in \text{poss } v$ **using** *poss-eq* **by** *auto*
have $u\text{-mtp}: \text{mtp-of } u \ p = \text{subst-mty } \varrho \ (\text{mtp-of } v \ p)$
using *mtp-of-subst*[*OF pv*] *u-eq* **by** *auto*
have $u'\text{-mtp}: \text{mtp-of } u' \ p = \text{subst-mty } \varrho' \ (\text{mtp-of } v \ p)$
using *mtp-of-subst*[*OF pv*] *u'-eq* **by** *auto*

Since v is an F-term, $\exists \sigma. \text{mtp-of } v \ p = \text{Some } \sigma$.

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

Need to show $\text{subst-ty } \varrho \ \sigma = \text{subst-ty } \varrho' \ \sigma$. By ($\text{subst-ty } ?\varrho \ ?\tau = \text{subst-ty } ?\varrho' \ ?\tau$) = $(\forall \alpha \in \text{tvars-ty } ?\tau. ?\varrho \ \alpha = ?\varrho' \ \alpha)$, suffices to show $\forall \alpha \in \text{tvars-ty } \sigma. \varrho \ \alpha = \varrho' \ \alpha$.

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

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

have $\text{mtp-of } t \ q \sqsubseteq_m \text{mtp-of } u \ q$ **using** *compl-mtp-of*[*OF compl-u*].
moreover have $\text{mtp-of } t \ q \sqsubseteq_m \text{mtp-of } u' \ q$ **using** *compl-mtp-of*[*OF compl-u'*].

ultimately have $eq\text{-at-}q: \text{mtp-of } u \ q = \text{mtp-of } t \ q \wedge \text{mtp-of } u' \ q = \text{mtp-of } t \ q$
using $t\text{-q}$ **by** (*auto simp: mcompl-def*)
then have $\text{mtp-of } u \ q = \text{mtp-of } u' \ q$ **by** *simp*

Now use $?p \in \text{poss } ?t \implies \text{mtp-of } (\text{subst-trm } ?\varrho \ ?t) \ ?p = \text{subst-mty } ?\varrho \ (\text{mtp-of } ?t \ ?p)$ to relate to substitutions.

have $\text{subst-mty } \varrho \ (\text{mtp-of } v \ q) = \text{subst-mty } \varrho' \ (\text{mtp-of } v \ q)$
using $\langle \text{mtp-of } u \ q = \text{mtp-of } u' \ q \rangle$ *mtp-of-subst*[*OF qv*] *u-eq u'-eq* **by** *auto*
then show $\varrho \ \alpha = \varrho' \ \alpha$ **using** $\alpha\text{-q}$
by (*cases mtp-of v q*) (*auto simp: subst-ty-agree*)

qed

then show $\text{mtp-of } u \ p = \text{mtp-of } u' \ p$ **using** $u\text{-mtp } u'\text{-mtp } vsig$ **by** *simp*

qed

qed

Strict (\sqsubseteq) witnesses a position difference.

lemma *compl-strict-witness*:

assumes $s' \sqsubseteq s \ s' \neq s$

shows $\exists p \in \text{poss } s. \text{mtp-of } s' \ p = \text{None} \wedge \text{mtp-of } s \ p \neq \text{None}$

```

using assms
proof (induction s' s rule: compl.induct)
  case (compl-App s1 s1' s2 s2'  $\xi$   $\xi'$ )
  show ?case
  proof (cases s1 = s1')
    case True
    then show ?thesis
    proof (cases s2 = s2')
      case True
      with  $\langle s1 = s1' \rangle$  compl-App.prems compl-App.hyps
      show ?thesis by (auto simp: mcompl-def)
    next
    case False
    with compl-App.IH(2) obtain p where  $p \in \text{poss } s2' \text{ mtp-of } s2 \text{ } p = \text{None}$ 
mtp-of } s2' \text{ } p \neq \text{None}
    by auto
    then show ?thesis by (intro beI[of - 2 # -]) auto
    qed
  next
  case False
  with compl-App.IH(1) obtain p where  $p \in \text{poss } s1' \text{ mtp-of } s1 \text{ } p = \text{None}$ 
mtp-of } s1' \text{ } p \neq \text{None}
  by auto
  then show ?thesis by (intro beI[of - Suc 0 # p]) auto
  qed
next
case (compl-Lam  $\zeta$   $\zeta'$  t t'  $\xi$   $\xi'$  x)
show ?case
proof (cases t = t')
  case True
  then show ?thesis
  proof (cases  $\zeta = \zeta'$ )
    case True
    with  $\langle t = t' \rangle$  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!: beI[of - [Suc 0]])
  qed
next
case False
with compl-Lam.IH obtain p where  $p \in \text{poss } t' \text{ mtp-of } t \text{ } p = \text{None}$ 
mtp-of } t' \text{ } p \neq \text{None}
by auto
then show ?thesis by auto
qed
qed (auto simp: mcompl-def)

```

Multiple completions from agreeing substitutions.

```

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

```

```

    using Ct.premis(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.premis(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.premis(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.premis(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.premis(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.premis(1) obtain s0 ζs ξs where
  seq: s = Lam x ζs s0 ξs
  s0 ⊆ subst-trm ρ t ζs ⊆m subst-mty ρ ξ ζs ⊆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  $\alpha$   $p$  assume  $p \in \text{poss } t \ \alpha \in \text{tvars-nty} \ (\text{mtp-of } t \ p) \ \text{mtp-of } s0 \ p \neq \text{None}$ 
    then show  $\varrho \ \alpha = \varrho' \ \alpha$  using  $\text{Lam.prem}(2)[\text{of } 2 \ \# \ p \ \alpha] \ \text{seq}(1)$  by auto
qed
moreover have  $\zeta s \sqsubseteq_m \text{subst-nty } \varrho' \ \xi$ 
proof (cases  $\xi$ )
  case None then show ?thesis using  $\text{seq}(3)$  by auto
next
  case (Some  $\sigma$ )
  show ?thesis
  proof (cases  $\zeta s$ )
    case None then show ?thesis by auto
  next
    case (Some  $\tau$ )
    then have  $\tau = \text{subst-ty } \varrho \ \sigma$  using  $\text{seq}(3) \ \langle \xi = \text{Some } \sigma \rangle$  by (auto simp:
mcompl-def)
    have  $\forall \alpha' \in \text{tvars-ty } \sigma. \ \varrho \ \alpha' = \varrho' \ \alpha'$ 
    proof (intro ballI)
      fix  $\alpha' \in \text{tvars-ty } \sigma$ 
      then show  $\varrho \ \alpha' = \varrho' \ \alpha'$ 
      using  $\text{Lam.prem}(2)[\text{of } [\text{Suc } 0] \ \alpha'] \ \langle \xi = \text{Some } \sigma \rangle \ \text{seq}(1) \ \text{Some}$  by auto
    qed
    then have  $\text{eq: } \text{subst-ty } \varrho \ \sigma = \text{subst-ty } \varrho' \ \sigma$  by (intro subst-ty-cong) auto
    from  $\langle \tau = \text{subst-ty } \varrho \ \sigma \rangle \ \text{eq}$  have  $\tau = \text{subst-ty } \varrho' \ \sigma$  by simp
    then show ?thesis using  $\text{Some } \langle \xi = \text{Some } \sigma \rangle$  by (auto simp: mcompl-def)
  qed
qed
moreover have  $\xi s \sqsubseteq_m \text{subst-nty } \varrho' \ \zeta$ 
proof (cases  $\zeta$ )
  case None then show ?thesis using  $\text{seq}(4)$  by auto
next
  case (Some  $\sigma$ )
  show ?thesis
  proof (cases  $\xi s$ )
    case None then show ?thesis by auto
  next
    case (Some  $\tau$ )
    then have  $\tau = \text{subst-ty } \varrho \ \sigma$  using  $\text{seq}(4) \ \langle \zeta = \text{Some } \sigma \rangle$  by (auto simp:
mcompl-def)
    moreover have  $\forall \alpha \in \text{tvars-ty } \sigma. \ \varrho \ \alpha = \varrho' \ \alpha$ 
    using  $\text{Lam.prem}(2)[\text{of } []] \ \text{seq}(1) \ \text{Some } \langle \zeta = \text{Some } \sigma \rangle$  by auto
    then have  $\text{subst-ty } \varrho \ \sigma = \text{subst-ty } \varrho' \ \sigma$  by (intro subst-ty-cong) auto
    ultimately show ?thesis using  $\text{Some } \langle \zeta = \text{Some } \sigma \rangle$  by (auto simp: mcompl-def)
  qed
qed
ultimately show ?case using  $\text{seq}(1)$  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**
 $fvars\ (Vr\ x\ (Some\ \sigma)) = \{(x, \sigma)\}$
 $| fvars\ (Vr\ x\ None) = \{\}$
 $| fvars\ (Ct\ c\ \xi) = \{\}$
 $| fvars\ (App\ t1\ t2\ \xi) = fvars\ t1 \cup fvars\ t2$
 $| fvars\ (Lam\ x\ \xi\ t\ \zeta) = \{(y, \sigma) \in fvars\ t.\ y \neq x\}$

lemma $fvars\ subst: fvars\ (subst-trm\ \rho\ t) = (\lambda(x, \sigma). (x, subst-ty\ \rho\ \sigma))\ `fvars\ t$
proof (*induction t*)

case $(Vr\ x\ \xi)$ **then show** *?case by (cases ξ) auto*
next
case $(Ct\ c\ \xi)$ **then show** *?case by auto*
next
case $(App\ t1\ t2\ \xi)$ **then show** *?case by auto*
next
case $(Lam\ x\ \xi\ t\ \zeta)$ **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**
 $wt-Vr: \vdash Vr\ x\ (Some\ \sigma)$
 $| wt-Ct: \exists \rho. \sigma = subst-ty\ \rho\ (ctpOf\ c) \Longrightarrow \vdash Ct\ c\ (Some\ \sigma)$
 $| wt-App: [\vdash u; \vdash v; tpOfR\ u = Arrow\ (tpOfR\ v)\ \sigma] \Longrightarrow \vdash App\ u\ v\ (Some\ \sigma)$
 $| wt-Lam: [\vdash u; \forall \tau. (x, \tau) \in fvars\ u \longrightarrow \tau = \sigma] \Longrightarrow \vdash Lam\ x\ (Some\ \sigma)\ u\ (Some\ (Arrow\ \sigma\ (tpOfR\ u)))$

Inversion rules for wt .

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

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

lemma *wt-App-inv*: $\vdash \text{App } t1 \ t2 \ (\text{Some } \sigma) \implies \vdash t1 \wedge \vdash t2 \wedge \text{tpOfR } t1 = \text{Arrow} \ (\text{tpOfR } t2) \ \sigma$
by (*erule wt.cases*) *auto*

lemma *wt-Lam-inv*: $\vdash \text{Lam } x \ (\text{Some } \sigma) \ t \ (\text{Some } \tau) \implies \vdash t \wedge \tau = \text{Arrow } \sigma \ (\text{tpOfR } t)$
by (*erule wt.cases*) *auto*

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

lemma *wt-Lam-invE*:
assumes $\vdash \text{Lam } x \ \xi \ u \ \zeta$
obtains σ **where** $\vdash u \ \xi = \text{Some } \sigma \ \zeta = \text{Some} \ (\text{Arrow } \sigma \ (\text{tpOfR } u))$
 $\forall \tau. (x, \tau) \in \text{fvvars } u \longrightarrow \tau = \sigma$
using *assms* **by** (*auto elim: wt.cases*)

lemma *wt-fterm*: $\vdash u \implies \text{fterm } u$
by (*induction u rule: wt.induct*) *auto*

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

lemma *subst-wt*:
assumes $\vdash v$
shows $\vdash \text{subst-trm } \rho \ v$
using *assms*
proof (*induction v rule: wt.induct*)
case (*wt-Vr x sigma*)
show *?case* **by** (*simp, rule wt.wt-Vr*)
next
case (*wt-Ct sigma c*)
then obtain $\rho\theta$ **where** $\sigma = \text{subst-ty } \rho\theta \ (\text{ctpOf } c)$ **by** *auto*
then have $\text{subst-ty } \rho \ \sigma = \text{subst-ty} \ (\rho \circ \rho\theta) \ (\text{ctpOf } c)$
by (*simp add: subst-ty-comp*)
then show *?case* **by** (*simp, intro wt.wt-Ct, auto*)
next
case (*wt-App u v sigma*)
from *wt-App* **have** *fu*: $\text{fterm } u \ \text{fterm } v$ **by** (*auto dest: wt-fterm*)
then obtain $\tau1 \ \tau2$ **where** $\text{mtp-of } u \ [] = \text{Some } \tau1 \ \text{mtp-of } v \ [] = \text{Some } \tau2$
using *fterm-mtp-of-Some* **by** (*metis nil-in-poss*)
with *wt-App* **have** *tp*: $\text{tpOfR} \ (\text{subst-trm } \rho \ u) = \text{Arrow} \ (\text{tpOfR} \ (\text{subst-trm } \rho \ v))$
 $(\text{subst-ty } \rho \ \sigma)$
by (*simp add: mtp-of-subst[of [] u rho, simplified] mtp-of-subst[of [] v rho, simplified]*)
from *wt-App tp* **show** *?case* **by** (*simp, intro wt.wt-App*) *auto*
next

```

case (wt-Lam u x  $\sigma$ )
from wt-Lam have fu: fterm u by (auto dest: wt-fterm)
then obtain  $\tau$  where mt: mtp-of u [] = Some  $\tau$ 
  using fterm-mtp-of-Some by (metis nil-in-poss)
from wt-Lam mt show ?case
proof -
  have eq: mtp-of (subst-trm  $\varrho$  u) [] = Some (subst-ty  $\varrho$   $\tau$ )
    using mt by (simp add: mtp-of-subst[of [] u  $\varrho$ , simplified])
  have tpR: tpOfR (subst-trm  $\varrho$  u) = subst-ty  $\varrho$   $\tau$  using eq by simp
  have tpU: tpOfR u =  $\tau$  using mt by simp
  have fv-cond:  $\forall \tau'. (x, \tau') \in \text{fvars} (\text{subst-trm } \varrho \text{ } u) \longrightarrow \tau' = \text{subst-ty } \varrho \text{ } \sigma$ 
    using wt-Lam(2) by (auto simp: fvars-subst)
  from wt-Lam have  $\vdash \text{Lam } x (\text{Some } (\text{subst-ty } \varrho \text{ } \sigma)) (\text{subst-trm } \varrho \text{ } u)$ 
    ( $\text{Some } (\text{Arrow } (\text{subst-ty } \varrho \text{ } \sigma) (\text{tpOfR } (\text{subst-trm } \varrho \text{ } u))))$ 
    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  $\sqsubseteq$  u  $\vdash$  u t  $\sqsubseteq$  v  $\vdash$  v
shows u = v
using assms
proof (induction t arbitrary: u v)
  case (Vr x  $\xi$ )
    then show ?case by (auto elim!: compl-Vr-leftE simp: mcompl-def)
  next
    case (Ct c  $\xi$ )
      then show ?case by (auto elim!: compl-Ct-leftE simp: mcompl-def)
  next
    case (App t1 t2  $\xi$ )
      from App.prem1 have ct1: cterm t1 cterm t2  $\xi$  = None by auto
      from App.prem2 have ua1: unambiguous t1 unambiguous t2 by auto
      from App.prem3(3) obtain u1 u2  $\xi u$  where ueq: u = App u1 u2  $\xi u$ 
        t1  $\sqsubseteq$  u1 t2  $\sqsubseteq$  u2  $\xi \sqsubseteq_m \xi u$ 
        by (auto elim!: compl-App-leftE)
      from App.prem5(5) obtain v1 v2  $\xi v$  where veq: v = App v1 v2  $\xi v$ 
        t1  $\sqsubseteq$  v1 t2  $\sqsubseteq$  v2  $\xi \sqsubseteq_m \xi v$ 
        by (auto elim!: compl-App-leftE)
      from App.prem4(4) ueq(1) obtain  $\sigma u$  where
        wt1:  $\vdash u1 \vdash u2$  and xiu:  $\xi u = \text{Some } \sigma u$  and
        tpu: tpOfR u1 = Arrow (tpOfR u2)  $\sigma u$ 
        by (auto elim!: wt-App-invE)
      from App.prem6(6) veq(1) obtain  $\sigma v$  where
        wt2:  $\vdash v1 \vdash v2$  and xiv:  $\xi v = \text{Some } \sigma v$  and
        tpv: tpOfR v1 = Arrow (tpOfR v2)  $\sigma v$ 

```

```

    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  $\xi$  t  $\zeta$ )
from Lam.prem1 have ct: cterm t  $\xi \neq \text{None}$   $\zeta = \text{None}$  by auto
from Lam.prem2 have ua: unambiguous t by auto
obtain  $\sigma$  where sig:  $\xi = \text{Some } \sigma$  using ct(2) by (cases  $\xi$ ) auto
from Lam.prem3(3) sig obtain u'  $\xi u$  where ueq:  $u = \text{Lam } x (\text{Some } \sigma) u' \xi u$ 
t  $\sqsubseteq$  u'
by (auto elim!: compl-Lam-leftE simp: mcompl-def)
from Lam.prem5(5) sig obtain v'  $\xi v$  where veq:  $v = \text{Lam } x (\text{Some } \sigma) v' \xi v$ 
t  $\sqsubseteq$  v'
by (auto elim!: compl-Lam-leftE simp: mcompl-def)
from Lam.prem4(4) ueq(1) obtain  $\tau u$  where
wtu:  $\vdash u'$  and xiusig:  $\text{Some } \sigma = \text{Some } \sigma$  and xiu:  $\xi u = \text{Some } (\text{Arrow } \sigma (\text{tpOfR } u'))$ 
by (auto elim!: wt-Lam-invE)
from Lam.prem6(6) veq(1) obtain  $\tau v$  where
wtv:  $\vdash v'$  and xivsig:  $\text{Some } \sigma = \text{Some } \sigma$  and xiv:  $\xi v = \text{Some } (\text{Arrow } \sigma (\text{tpOfR } v'))$ 
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* :: $(\text{'tv}, \text{'k}, \text{'v}, \text{'c}) \text{ trm} \Rightarrow (\text{'tv}, \text{'k}, \text{'v}, \text{'c}) \text{ trm} \Rightarrow \text{bool}$ **where**
is-wt-completion t u \longleftrightarrow t \sqsubseteq u \wedge fterm u \wedge $\vdash u$

definition *is-mgen* :: $(\text{'tv}, \text{'k}, \text{'v}, \text{'c}) \text{ trm} \Rightarrow (\text{'tv}, \text{'k}, \text{'v}, \text{'c}) \text{ trm} \Rightarrow \text{bool}$ **where**
is-mgen t v \longleftrightarrow *is-wt-completion* t v \wedge
 $(\forall u. \text{is-wt-completion } t u \longrightarrow (\exists \varrho. u = \text{subst-trm } \varrho v))$

definition *typeable* :: $(\text{'tv}, \text{'k}, \text{'v}, \text{'c}) \text{ trm} \Rightarrow \text{bool}$ **where**
typeable t \longleftrightarrow $(\exists u. \text{is-wt-completion } t u)$

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* :: $\text{'c} \Rightarrow (\text{'tv}, \text{'k}) \text{ ty} +$

assumes *mgen-exists*:
 $\llbracket \text{typeable } (t :: ('tv, 'k, 'v, 'c) \text{ trm}); \text{unambiguous } t \rrbracket \implies \exists v. \text{is-mgen } t \ v$ **and**
tyvars-inf: *infinite* (*UNIV* :: *'tv set*)

begin

definition *mgen* :: $('tv, 'k, 'v, 'c) \text{ trm} \Rightarrow ('tv, 'k, 'v, 'c) \text{ trm}$ **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** $\exists v. \text{is-mgen } t \ v$
by (*intro mgen-exists*)

then show *?thesis* **unfolding** *mgen-def* **by** (*rule someI-ex*)

qed

lemma *mgen-compl*:
assumes *typeable* *t* *unambiguous* *t*
shows $t \sqsubseteq \text{mgen } t$
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 $\vdash \text{mgen } t$
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 $\exists \varrho. u = \text{subst-trm } \varrho \ (\text{mgen } t)$
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 $\langle \text{correct-printing } t \ s \longleftrightarrow (\forall t'. \text{is-mgen } s \ t' \longleftrightarrow t' = \text{mgen } t) \rangle$

definition $\langle \text{strong-correct-printing } t \ s \longleftrightarrow (\forall t'. \text{is-wt-completion } s \ t' \longleftrightarrow t' = \text{mgen } t) \rangle$

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 $\neq \{\}$
shows $\exists v'. v' \neq v \wedge \text{is-mgen } s v'$
proof –
from *mg* **have** *sv*: $s \sqsubseteq v$ **and** *fv*: *fterm v* **and** *wtv*: $\vdash v$
and *mg-prop*: $\bigwedge u. \text{is-wt-completion } s u \implies \exists \varrho. u = \text{subst-trm } \varrho v$
by (*auto simp*: *is-mgen-def is-wt-completion-def*)
from *extra* **obtain** α **where** *alpha-v*: $\alpha \in \text{tvars-trm } v$ **and** *alpha-ns*: $\alpha \notin \text{tvars-trm } s$
by *auto*

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

have *prop-a*: *mtp-of s p* = *None* **if** $p \in \text{poss } v \alpha \in \text{tvars-mty } (\text{mtp-of } v p)$ **for** *p*
proof (*rule ccontr*)
assume *mtp-of s p* $\neq \text{None}$
have $p \in \text{poss } s$ **using** *that(1)* *compl-poss-eq[OF sv]* **by** *simp*
have *mtp-of s p* \sqsubseteq_m *mtp-of v p* **using** *compl-mtp-of[OF sv]* .
then **have** *mtp-of s p* = *mtp-of v p* **using** $\langle \text{mtp-of } s p \neq \text{None} \rangle$ **by** (*auto simp*:
mcompl-def)
then **have** $\alpha \in \text{tvars-mty } (\text{mtp-of } s p)$ **using** *that(2)* **by** *simp*
then **have** $\alpha \in \text{tvars-trm } s$ **using** *tvars-trm-eq-UN-mtp* $\langle p \in \text{poss } s \rangle$ **by** *fastforce*
with *alpha-ns* **show** *False* **by** *contradiction*
qed
obtain β **where** *beta-nv*: $\beta \notin \text{tvars-trm } v$
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*

$v \models v'$.

have *v-neq*: $v \neq v'$
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* = *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'*: $\vdash v'$ **unfolding** *v'-def* **using** *subst-wt[OF wtv]* .
have *fv'*: *fterm v'* **unfolding** *v'-def* **using** *fv* **by** (*induction v*) *auto*

$s \sqsubseteq v'$ **using** $\llbracket ?s \sqsubseteq \text{subst-trm } ?\varrho ?v; \bigwedge \alpha p. \llbracket p \in \text{poss } ?v; \alpha \in \text{tvars-mty } (\text{mtp-of } ?v p); \text{mtp-of } ?s p \neq \text{None} \rrbracket \implies ?\varrho \alpha = ?\varrho' \alpha \rrbracket \implies ?s \sqsubseteq \text{subst-trm } ?\varrho' ?v.$

have *sv'*: $s \sqsubseteq v'$
proof –
have $s \sqsubseteq \text{subst-trm } \text{TyVar } v$ **using** *sv* **by** *simp*
then **show** $s \sqsubseteq v'$ **unfolding** *v'-def*
proof (*rule multiple-compl*)
fix $\alpha' p$ **assume** $p \in \text{poss } v \alpha' \in \text{tvars-mty } (\text{mtp-of } v p) \text{mtp-of } s p \neq \text{None}$
then **have** $\alpha \notin \text{tvars-mty } (\text{mtp-of } v p)$ **using** *prop-a* **by** *auto*

then have $\alpha' \neq \alpha$ **using** $\langle \alpha' \in \text{tvars-nty} (\text{mtp-of } v \text{ } p) \rangle$ **by** *auto*
then show $\text{TyVar } \alpha' = \text{single-subst } (\text{TyVar } \beta) \alpha \alpha'$ **by** (*simp add: single-subst-def*)
qed
qed

v' is most general.

have $\bigwedge u. \text{is-wt-completion } s \ u \implies \exists \varrho. u = \text{subst-trm } \varrho \ v'$
proof –
fix u **assume** $\text{is-wt-completion } s \ u$
then obtain ϱ **where** $u = \text{subst-trm } \varrho \ v$ **using** *mg-prop* **by** *auto*
have $v = \text{subst-trm } (\text{single-subst } (\text{TyVar } \alpha) \beta) \ v'$
unfolding v' -*def* **using** *subst-trm-swap[OF beta-nv]* **by** *simp*
then have $u = \text{subst-trm } (\varrho \circ \text{single-subst } (\text{TyVar } \alpha) \beta) \ v'$
using $\langle u = \text{subst-trm } \varrho \ v \rangle$ *subst-trm-comp* **by** *metis*
then show $\exists \varrho. u = \text{subst-trm } \varrho \ v'$ **by** *blast*
qed
then have $\text{is-mgen } s \ v'$
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 $s :: \langle ('tv, 'k, 'v, 'c) \text{ trm} \rangle$
assumes $\text{unambiguous } s \ \text{is-wt-completion } s \ u \ \text{is-wt-completion } s \ u' \ u \neq u'$
obtains $v \ v'$ **where** $\langle v \neq v' \rangle \langle \text{is-mgen } s \ v \rangle \langle \text{is-mgen } s \ v' \rangle$
proof –
have ty - s : *typeable* s **using** *assms(2)* **by** (*auto simp: typeable-def*)
have mg : $\text{is-mgen } s \ (\text{mgen } s)$ **using** *mgen-is-mgen[OF ty-s assms(1)]* .
obtain ϱ **where** u -*eq*: $u = \text{subst-trm } \varrho \ (\text{mgen } s)$
using *mgen-most-general[OF ty-s assms(1,2)]* **by** *auto*
obtain ϱ' **where** u' -*eq*: $u' = \text{subst-trm } \varrho' \ (\text{mgen } s)$
using *mgen-most-general[OF ty-s assms(1,3)]* **by** *auto*
from *assms(4)* u -*eq* u' -*eq* **have** $\exists \alpha \in \text{tvars-trm } (\text{mgen } s). \varrho \ \alpha \neq \varrho' \ \alpha$
by (*auto simp: subst-trm-agree*)
then obtain α **where** α - v : $\alpha \in \text{tvars-trm } (\text{mgen } s)$ **and** ρ - neq : $\varrho \ \alpha \neq \varrho' \ \alpha$
 α **by** *auto*
have sv : $s \sqsubseteq \text{mgen } s$ **using** *mgen-compl[OF ty-s assms(1)]* .
have $\alpha \notin \text{tvars-trm } s$
proof
assume $\alpha \in \text{tvars-trm } s$
then obtain p **where** ps : $p \in \text{poss } s$ **and** α - s : $\alpha \in \text{tvars-nty} (\text{mtp-of } s \ p)$
using *tvars-trm-eq-UN-mtp* **by** *fastforce*
then have $snone$: $\text{mtp-of } s \ p \neq \text{None}$ **by** (*cases mtp-of s p*) *auto*
have pv : $p \in \text{poss } (\text{mgen } s)$ **using** ps *compl-poss-eq[OF sv]* **by** *simp*
have $\text{mtp-of } s \ p = \text{mtp-of } (\text{mgen } s) \ p$
using *compl-mtp-of[OF sv, of p] snone* **by** (*auto simp: mcompl-def*)
then have α - vp : $\alpha \in \text{tvars-nty} (\text{mtp-of } (\text{mgen } s) \ p)$ **using** α - 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 \Rightarrow ('tv,'k,'v,'c) trm **where**
smobla t = decrease (mgen (erase t)) (mgen t)

1.7 Completeness

Auxiliary: decrease only removes annotations, so $decrease\ v\ s \sqsubseteq s$.

lemma decrease-compl:

decrease v s \sqsubseteq 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: $erase\ t \sqsubseteq decrease\ v\ s$ if $erase\ t \sqsubseteq s$.

lemma erase-compl-decrease:

erase t \sqsubseteq s \implies erase t \sqsubseteq 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 \sqsubseteq subst-trm ρ v

$\forall \alpha \in \text{tvars-trm } v. \exists p \in \text{poss } v. \alpha \in \text{tvars-mty } (\text{mtp-of } v\ p) \wedge \text{mtp-of } s\ p \neq$

None

shows $\forall \alpha \in \text{tvars-trm } v. \exists p \in \text{poss } v. \alpha \in \text{tvars-mty } (\text{mtp-of } v\ p) \wedge$

$\text{mtp-of } (\text{decrease } v\ s)\ p \neq \text{None}$

using assms(2,3)

proof (induction v s rule: decrease.induct)

case (1 v s)

show ?case

proof (cases $\exists p. \text{test } v\ s\ p$)

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

```

$q = \text{pickPos } v \ s$, so $\text{mtp-of } s \ q$ 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*: \langle *is-wt-completion ?s u* \rangle **for** *u*
proof –
from *u-wc* **have** *su*: *?s* \sqsubseteq *u* **and** *fu*: *fterm u* **and** *wtu*: \vdash *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*: $\forall \alpha \in$ *tvars-trm ?v*. $\exists p \in$ *poss ?v*. $\alpha \in$ *tvars-mty (mtp-of ?v p)* \wedge
mtp-of (mgen t) p \neq *None*
proof (*intro ballI*)
fix α **assume** $\alpha \in$ *tvars-trm ?v*
then obtain *p* **where** $p \in$ *poss ?v* $\alpha \in$ *tvars-mty (mtp-of ?v p)*

using *tvars-trm-eq-UN-mtp* **by** *fastforce*
moreover have *mtp-of (mgen t) p ≠ None*
proof –
 have $p \in \text{poss } (mgen\ t)$ **using** $\langle p \in \text{poss } ?v \rangle$ *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 $\exists p \in \text{poss } ?v. \alpha \in \text{tvars-mty } (mtp\text{-of } ?v\ p) \wedge mtp\text{-of } (mgen\ t)\ p \neq \text{None}$
 by *auto*
 qed
 have *s-compl-subst: ?s \sqsubseteq subst-trm ρ ?v* **using** *s-compl-mgen mgen-inst* **by**
simp
 have *cov-mgen: $\forall \alpha \in \text{tvars-trm } ?v. \exists p \in \text{poss } ?v. \alpha \in \text{tvars-mty } (mtp\text{-of } ?v\ p)$*
 \wedge
 mtp-of (mgen t) p ≠ None **using** *cov* .
 have *cov-s: $\forall \alpha \in \text{tvars-trm } ?v. \exists p \in \text{poss } ?v. \alpha \in \text{tvars-mty } (mtp\text{-of } ?v\ p) \wedge mtp\text{-of } ?s\ p \neq \text{None}$*
proof (*intro ballI*)
 fix α **assume** $\alpha \in \text{tvars-trm } ?v$
 have *dec-cov: $\forall \alpha \in \text{tvars-trm } ?v. \exists p \in \text{poss } ?v. \alpha \in \text{tvars-mty } (mtp\text{-of } ?v\ p)$*
 \wedge
 mtp-of (decrease ?v (mgen t)) p ≠ None
proof –
 have *mgen t \sqsubseteq subst-trm ρ ?v* **using** *mgen-inst* **by** *simp*
 then show *?thesis* **using** *decrease-coverage*[*OF fv*] *cov-mgen* **by** *blast*
 qed
 with $\langle \alpha \in \text{tvars-trm } ?v \rangle$ **obtain** p **where** $p \in \text{poss } ?v$
 $\alpha \in \text{tvars-mty } (mtp\text{-of } ?v\ p) \wedge mtp\text{-of } (decrease\ ?v\ (mgen\ t))\ p \neq \text{None}$
 by *auto*
 then show $\exists p \in \text{poss } ?v. \alpha \in \text{tvars-mty } (mtp\text{-of } ?v\ p) \wedge mtp\text{-of } ?s\ p \neq \text{None}$
 unfolding *s-def* **by** *blast*
 qed

Apply $\llbracket ?t \sqsubseteq ?u; fterm\ ?u; \exists \rho. ?u = \text{subst-trm } \rho\ ?v; fterm\ ?v; \forall \alpha \in \text{tvars-trm } ?v. \exists p \in \text{poss } ?v. \alpha \in \text{tvars-mty } (mtp\text{-of } ?v\ p) \wedge mtp\text{-of } ?t\ p \neq \text{None}; ?t \sqsubseteq ?u'; fterm\ ?u'; \exists \rho'. ?u' = \text{subst-trm } \rho'\ ?v \rrbracket \implies ?u = ?u'$.

have $\exists \rho. mgen\ t = \text{subst-trm } \rho\ ?v$ **using** *mgen-inst* **by** *auto*
moreover have $\exists \rho'. u = \text{subst-trm } \rho'\ ?v$ **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*]
 $\langle \exists \rho. mgen\ t = \text{subst-trm } \rho\ ?v \rangle$ *fv cov-s su fu* $\langle \exists \rho'. u = \text{subst-trm } \rho'\ ?v \rangle$])
 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:
   $\neg \text{test } v \text{ (decrease } v \text{ s) } p$ 
proof (induction v s rule: decrease.induct)
  case (1 v s)
  show ?case
  proof (cases  $\exists p. \text{test } v \text{ s } p$ )
    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:
   $\text{unambiguous } s \implies \text{unambiguous (decrease } v \text{ s)}$ 
proof (induction v s rule: decrease.induct)
  case (1 v s)
  show ?case
  proof (cases  $\exists p. \text{test } v \text{ s } p$ )
    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: $s' \sqsubseteq \text{smobla } t$ **and** s' -neg: $s' \neq \text{smobla } t$
obtains $v v'$ **where** $v \neq v'$ *is-mgen* $s' v$ *is-mgen* $s' v'$
proof –
let $?s = \text{smobla } t$
let $?v = \text{mgen } (\text{erase } t)$
have $s\text{-}ua$: 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 $\text{smobla } t$ has non-*None*.

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

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

have *no-test*: $\neg \text{test } ?v ?s p$ **using** *decrease-no-test* **unfolding** *smobla-def* **by** *blast*

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

have $s\text{-}compl\text{-}mgen$: $?s \sqsubseteq \text{mgen } t$
using *decrease-compl smobla-def* **by** *metis*
have $s'\text{-}compl\text{-}mgen$: *is-wt-completion* s' (*mgen* t)
using *compl-trans*[OF s' -compl $s\text{-}compl\text{-}mgen$] *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 $er\text{-}ty$: typeable (*erase* t)
using ty **by** (*auto simp: typeable-def is-wt-completion-def intro: compl-trans*[OF *erase-compl*])
have $er\text{-}ua$: unambiguous (*erase* t) **using** ua **by** *simp*
obtain ϱ **where** $mgen\text{-}inst$: $mgen } t = \text{subst-trm } \varrho ?v$
using *mgen-most-general*[OF $er\text{-}ty$ $er\text{-}ua$]
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 $er\text{-}ty$ $er\text{-}ua$] .
have wtv : $\vdash ?v$ **using** *mgen-wt*[OF $er\text{-}ty$ $er\text{-}ua$] .
have $poss\text{-}sv$: $\text{poss } ?s = \text{poss } ?v$
using *compl-poss-eq*[OF $s\text{-}compl\text{-}mgen$] $mgen\text{-}inst$ **by** *simp*
have pv : $p \in \text{poss } ?v$ **using** ps $poss\text{-}sv$ **by** *simp*

Extract tyvar from the negated test.

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

Property (c): α is uncovered in s' .

have *poss-s': $\text{poss } s' = \text{poss } ?s$* **using** *compl-poss-eq[OF s'-compl]* .
have *prop-c: $\langle \text{mtp-of } s' q = \text{None} \rangle$* **if** $\langle q \in \text{poss } s' \rangle$ **and** $\langle \alpha \in \text{tvars-mty} (\text{mtp-of } ?v q) \rangle$ **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: $s' \sqsubseteq \text{subst-trm } \varrho \ ?v$*
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 $w :: ('tv, 'k) \text{ ty}$ **where** *w-neq: $w \neq \varrho \ \alpha$*
using *ty.distinct(1)* **by** *metis*
define ϱ' **where** $\varrho' = (\lambda \beta. \text{if } \beta = \alpha \text{ then } w \text{ else } \varrho \ \beta)$

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

have *agree: $\varrho \ \beta = \varrho' \ \beta$* **if** $q \in \text{poss } ?v \ \beta \in \text{tvars-mty} (\text{mtp-of } ?v q) \ \text{mtp-of } s' q \neq \text{None}$
for $\beta \ q$ **using** *that* ϱ' -def *poss-s' poss-sv prop-c* **by** *auto*
have *s'-compl-vrho': $s' \sqsubseteq \text{subst-trm } \varrho' \ ?v$*
using *multiple-compl[OF s'-compl-vrho agree]* .
have *wtrho': $\vdash \text{subst-trm } \varrho' \ ?v$* **using** *subst-wt[OF wtv]* .
have $\alpha \in \text{tvars-trm } ?v$ **using** *alpha-vp tvars-trm-eq-UN-mtp pv* **by** *fastforce*
then have *vrho-neq: $\text{subst-trm } \varrho \ ?v \neq \text{subst-trm } \varrho' \ ?v$*
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 wtrho'* **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 $\langle s \sqsubseteq \text{smobla } t \rangle$ $\langle \text{correct-printing } t \ s \rangle$*
shows $\langle s = \text{smobla } t \rangle$
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  $\Rightarrow$  ty  $\Rightarrow$  ty (infixr  $\rightarrow$  65) where  
  fun-ty  $\tau_1$   $\tau_2$  = TCons "fun" [ $\tau_1$ ,  $\tau_2$ ]
```

Type variables occurring in a type.

```
fun tvars-ty :: ty  $\Rightarrow$  string set where  
  tvars-ty (TVar v) = {v}  
| tvars-ty (TCons - ts) =  $\bigcup$ (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  $\Rightarrow$  ty)  $\Rightarrow$  ty  $\Rightarrow$  ty where  
  subst-ty  $\sigma$  (TVar v) =  $\sigma$  v  
| subst-ty  $\sigma$  (TCons k ts) = TCons k (map (subst-ty  $\sigma$ ) ts)
```

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

```
lemma arrow-neq-self:  $\tau \rightarrow \tau \neq \tau$ 
```

proof

assume $\tau \rightarrow \tau = \tau$
then have $\text{size } (\tau \rightarrow \tau) = \text{size } \tau$ **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*:

$\text{subst-ty } \sigma_1 \tau = \text{subst-ty } \sigma_2 \tau \implies \alpha \in \text{tvars-ty } \tau \implies \sigma_1 \alpha = \sigma_2 \alpha$
by (*induction* τ) *auto*

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

lemma *tvars-subst-ty*: $\text{tvars-ty } (\text{subst-ty } \sigma \tau) = \bigcup (\text{tvars-ty } \text{' } \sigma \text{' } \text{tvars-ty } \tau)$
by (*induction* τ) *auto*

lemma *subst-ty-id*:

$(\bigwedge v. v \in \text{tvars-ty } \tau \implies \sigma v = \text{TVar } v) \implies \text{subst-ty } \sigma \tau = \tau$
by (*induction* τ) (*auto intro: map-idI*)

lemma *subst-ty-agree*:

$(\bigwedge v. v \in \text{tvars-ty } \tau \implies \sigma_1 v = \sigma_2 v) \implies \text{subst-ty } \sigma_1 \tau = \text{subst-ty } \sigma_2 \tau$
by (*induction* τ) *auto*

lemma *subst-ty-compose*:

$\text{subst-ty } \sigma_1 (\text{subst-ty } \sigma_2 \tau) = \text{subst-ty } (\lambda v. \text{subst-ty } \sigma_1 (\sigma_2 v)) \tau$
by (*induction* τ) *auto*

2.2 Contexts

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

type-synonym $\text{ctx} = \text{string} \rightarrow \text{ty}$

Type variables of a context.

definition *tvars-ctx* :: $\text{ctx} \Rightarrow \text{string set}$ **where**

$\text{tvars-ctx } \Gamma = \bigcup (\text{tvars-ty } \text{' } \text{ran } \Gamma)$

Applying a type substitution to a context.

definition *subst-ctx* :: $(\text{string} \Rightarrow \text{ty}) \Rightarrow \text{ctx} \Rightarrow \text{ctx}$ **where**

$\text{subst-ctx } \sigma \Gamma = \text{map-option } (\text{subst-ty } \sigma) \circ \Gamma$

lemma *subst-ctx-dom* [*simp*]: $\text{dom } (\text{subst-ctx } \sigma \Gamma) = \text{dom } \Gamma$

by (*auto simp: subst-ctx-def dom-def*)

lemma *subst-ctx-app*: $x \in \text{dom } \Gamma \implies \text{subst-ctx } \sigma \Gamma x = \text{Some } (\text{subst-ty } \sigma (\text{the } (\Gamma x)))$

by (*auto simp: subst-ctx-def dom-def*)

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

lemma *subst-ctx-id*:
 $(\bigwedge v. v \in tvars-ctx\ \Gamma \implies \sigma\ v = TVar\ v) \implies subst-ctx\ \sigma\ \Gamma = \Gamma$
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* \Rightarrow *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* \Rightarrow *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 \wedge *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  $\Rightarrow$  ty where
  typeof-aterm (AConst -  $\tau$ ) =  $\tau$ 
| typeof-aterm (AVar -  $\tau$ ) =  $\tau$ 
| typeof-aterm (AAbs - - -  $\tau'$ ) =  $\tau'$ 
| typeof-aterm (AApp - -  $\tau$ ) =  $\tau$ 
```

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

```
fun erase :: aterm  $\Rightarrow$  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  $\Rightarrow$  ty)  $\Rightarrow$  aterm  $\Rightarrow$  aterm where
  subst-aterm  $\sigma$  (AConst c  $\tau$ ) = AConst c (subst-ty  $\sigma$   $\tau$ )
| subst-aterm  $\sigma$  (AVar x  $\tau$ ) = AVar x (subst-ty  $\sigma$   $\tau$ )
| subst-aterm  $\sigma$  (AAbs x  $\tau$  a  $\tau'$ ) = AAbs x (subst-ty  $\sigma$   $\tau$ ) (subst-aterm  $\sigma$  a) (subst-ty  $\sigma$   $\tau'$ )
| subst-aterm  $\sigma$  (AApp a1 a2  $\tau$ ) = AApp (subst-aterm  $\sigma$  a1) (subst-aterm  $\sigma$  a2) (subst-ty  $\sigma$   $\tau$ )
```

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* \Rightarrow *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* $\tau \cup \text{tvars-aterm } a \cup \text{tvars-ty } \tau'$
| *tvars-aterm* (*AApp* *a*₁ *a*₂ τ) = *tvars-aterm* *a*₁ \cup *tvars-aterm* *a*₂ \cup *tvars-ty* τ

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

lemma *subst-aterm-agree*:

($\bigwedge v. v \in \text{tvars-aterm } a \implies \sigma_1 v = \sigma_2 v$) $\implies \text{subst-aterm } \sigma_1 a = \text{subst-aterm } \sigma_2 a$
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 $\sigma_1 a = \text{subst-aterm } \sigma_2 a \implies \alpha \in \text{tvars-aterm } a \implies \sigma_1 \alpha = \sigma_2 \alpha$
using *unique-type-match* **by** (*induction a arbitrary: α*) *auto*

lemma *subst-ty-TVar* [*simp*]: *subst-ty* *TVar* $\tau = \tau$

by (*induction τ*) (*auto simp: map-idI*)

lemma *subst-aterm-id*:

($\bigwedge v. v \in \text{tvars-aterm } a \implies \sigma v = \text{TVar } v$) $\implies \text{subst-aterm } \sigma a = a$
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* \Rightarrow *ty*) \Rightarrow *ctx* \Rightarrow *aterm* \Rightarrow *bool* **where**

wt-const: $\exists \varrho. \tau = \text{subst-ty } \varrho (\text{const-type } c) \implies$
 well-typed const-type Γ (*AConst* *c* τ)
| *wt-var*: $\Gamma x = \text{Some } \tau \implies$
 well-typed const-type Γ (*AVar* *x* τ)
| *wt-abs*: *well-typed const-type* ($\Gamma(x \mapsto \tau)$) *a* \implies
 $\tau' = \tau \rightarrow \text{typeof-aterm } a \implies$
 well-typed const-type Γ (*AAbs* *x* τ *a* τ')
| *wt-app*: *well-typed const-type* $\Gamma a_1 \implies$

$$\begin{aligned}
& \text{well-typed const-type } \Gamma \ a_2 \implies \\
& \text{typeof-aterm } a_1 = \text{typeof-aterm } a_2 \rightarrow \tau \implies \\
& \text{well-typed const-type } \Gamma \ (AApp \ a_1 \ a_2 \ \tau)
\end{aligned}$$

Lemma (Substitution Preserves Well-Typedness). If a is well-typed in Γ , then $\text{subst-aterm } \sigma \ a$ is well-typed in $\text{subst-ctx } \sigma \ \Gamma$.

lemma *subst-preserves-wt*:
 $\text{well-typed } ct \ \Gamma \ a \implies \text{well-typed } ct \ (\text{subst-ctx } \sigma \ \Gamma) \ (\text{subst-aterm } \sigma \ a)$
proof (*induction rule: well-typed.induct*)
case (*wt-const* $\tau \ ct \ c \ \Gamma$)
then obtain ϱ **where** $\tau = \text{subst-ty } \varrho \ (ct \ c)$ **by** *auto*
then have $\text{subst-ty } \sigma \ \tau = \text{subst-ty } (\lambda v. \text{subst-ty } \sigma \ (\varrho \ v)) \ (ct \ c)$
by (*simp add: subst-ty-compose*)
then show $?case$ **by** (*auto intro: well-typed.wt-const*)
next
case (*wt-var* $\Gamma \ x \ \tau \ ct$)
then show $?case$ **by** (*auto simp: subst-ctx-def intro: well-typed.wt-var*)
next
case (*wt-abs* $ct \ \Gamma \ x \ \tau \ a \ \tau'$)
then have ih : $\text{well-typed } ct \ ((\text{subst-ctx } \sigma \ \Gamma)(x \mapsto \text{subst-ty } \sigma \ \tau)) \ (\text{subst-aterm } \sigma \ a)$
using *subst-ctx-update* **by** *metis*
from *wt-abs.hyps(2)* **have** $\text{subst-ty } \sigma \ \tau' = \text{subst-ty } \sigma \ \tau \rightarrow \text{typeof-aterm } (\text{subst-aterm } \sigma \ a)$
by (*simp add: fun-ty-def*)
with ih **show** $?case$ **by** (*auto intro: well-typed.wt-abs*)
next
case (*wt-app* $ct \ \Gamma \ a_1 \ a_2 \ \tau$)
from *wt-app.hyps(3)* **have**
 $\text{typeof-aterm } (\text{subst-aterm } \sigma \ a_1) = \text{typeof-aterm } (\text{subst-aterm } \sigma \ a_2) \rightarrow \text{subst-ty } \sigma \ \tau$
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 $\text{subst-ctx } \sigma \ \Gamma = \Gamma$, so $\text{subst-aterm } \sigma \ a$ is well-typed in Γ .

corollary *subst-wt-identity-ctx*:
 $\text{well-typed } ct \ \Gamma \ a \implies (\bigwedge v. v \in \text{tvars-ctx } \Gamma \implies \sigma \ v = TVar \ v) \implies$
 $\text{well-typed } ct \ \Gamma \ (\text{subst-aterm } \sigma \ a)$
using *subst-preserves-wt[of ct $\Gamma \ a \ \sigma$ subst-ctx-id[of $\Gamma \ \sigma$]* **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* $:: nat \Rightarrow (nat \times raw-term \times ty) list \Rightarrow (nat \times raw-term \times ty) list$ **where**

$$shift-enum\ k\ L = map\ (\lambda(p, s, \tau). (k + p, s, \tau))\ L$$

fun *enum-aterm* $:: aterm \Rightarrow (nat \times raw-term \times ty) list$ **where**

$$\begin{aligned} &enum-aterm\ (AConst\ c\ \tau) = [(0, RConst\ c, \tau)] \\ | &enum-aterm\ (AVar\ x\ \tau) = [(0, RVar\ x, \tau)] \\ | &enum-aterm\ (AAbs\ x\ \tau\ a_1\ \tau') = \\ &\quad (let\ L = enum-aterm\ a_1; n = length\ L\ in \\ &\quad\ [(0, RVar\ x, \tau)]\ @\ shift-enum\ 1\ L\ @\ [(1 + n, RAbs\ x\ (erase\ a_1), \tau')]) \\ | &enum-aterm\ (AApp\ a_1\ a_2\ \tau) = \\ &\quad (let\ L_1 = enum-aterm\ a_1; n_1 = length\ L_1; \\ &\quad\ L_2 = enum-aterm\ a_2; n_2 = length\ L_2\ in \\ &\quad\ L_1\ @\ shift-enum\ n_1\ L_2\ @\ [(n_1 + n_2, RApp\ (erase\ a_1)\ (erase\ a_2), \tau)]) \end{aligned}$$

The set of positions.

definition *pos-set* $:: aterm \Rightarrow nat\ set$ **where**

$$pos-set\ a = fst\ 'set\ (enum-aterm\ a)$$

The type at a given position in the enumeration.

definition *type-at-pos* $:: aterm \Rightarrow nat \Rightarrow ty$ **where**

$$type-at-pos\ a\ p = (THE\ \tau.\ \exists s. (p, s, \tau) \in set\ (enum-aterm\ a))$$

The subterm at a given position in the enumeration.

definition *subterm-at-pos* $:: aterm \Rightarrow nat \Rightarrow raw-term$ **where**

$$subterm-at-pos\ a\ p = (THE\ s.\ \exists \tau. (p, s, \tau) \in set\ (enum-aterm\ a))$$

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*: $set\ (shift-enum\ k\ L) = (\lambda(p, s, \tau). (k + p, s, \tau))\ 'set\ L$
by (*auto simp: shift-enum-def*)

lemma *in-shift-enum-iff*:

$$(p, s, \tau) \in set\ (shift-enum\ k\ L) \iff (\exists p'. p = k + p' \wedge (p', s, \tau) \in set\ L)$$

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* $:: aterm \Rightarrow nat$ **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*: $enum-aterm a \neq []$
by (*cases a*) (*auto simp: Let-def*)

lemma *aterm-node-count-pos*: $aterm-node-count a \geq 1$
by (*cases a*) *auto*

2.4.2 Substitution Distributes to Positions

Lemma (Substitution Distributes to Positions). The enumeration of *subst-aterm* σa 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* ::
 $(string \Rightarrow ty) \Rightarrow (nat \times raw-term \times ty) list \Rightarrow (nat \times raw-term \times ty) list$ **where**
 $map-enum-ty \sigma L = map (\lambda(p, s, \tau). (p, s, subst-ty \sigma \tau)) L$

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

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

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

The main lemma: the enumeration commutes with type substitution.

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

corollary *type-at-pos-subst-iff*:
 $(p, s, \tau') \in set (enum-aterm (subst-aterm \sigma a)) \iff$
 $(\exists \tau. (p, s, \tau) \in set (enum-aterm a) \wedge \tau' = subst-ty \sigma \tau)$
by (*auto simp: enum-subst-aterm map-enum-ty-def image-iff split: prod.splits*)

corollary *type-at-pos-subst*:

$(p, s, \tau) \in \text{set } (\text{enum-aterm } a) \implies (p, s, \text{subst-ty } \sigma \tau) \in \text{set } (\text{enum-aterm } (\text{subst-aterm } \sigma a))$
using *type-at-pos-subst-iff* **by** *auto*

The node count is invariant under type substitution.

lemma *aterm-node-count-subst* [*simp*]:

$\text{aterm-node-count } (\text{subst-aterm } \sigma a) = \text{aterm-node-count } a$
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*]: $\text{pos-set } (\text{subst-aterm } \sigma a) = \text{pos-set } a$

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*:

$\alpha \in \text{tvars-aterm } a \implies \exists p s \tau. (p, s, \tau) \in \text{set } (\text{enum-aterm } a) \wedge \alpha \in \text{tvars-ty } \tau$
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*:

$(p, s, \tau) \in \text{set } (\text{enum-aterm } a) \implies \text{tvars-ty } \tau \subseteq \text{tvars-aterm } a$
by (*induction a arbitrary: p s \tau*) (*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*: $\text{map fst } (\text{shift-enum } k L) = \text{map } ((+) k) (\text{map fst } L)$
by (*induction L*) (*auto simp: shift-enum-def*)

lemma *map-fst-enum-aterm*: $\text{map fst } (\text{enum-aterm } a) = [0..<\text{aterm-node-count } a]$

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*: $\text{distinct } (\text{map fst } (\text{enum-aterm } a))$

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

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

lemma *pos-set-range*: $\text{pos-set } a = \{0..<\text{aterm-node-count } a\}$

unfolding *pos-set-def*

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

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

lemma *enum-aterm-unique*:

assumes $(p, s, \tau) \in \text{set } (\text{enum-aterm } a)$ $(p, s', \tau') \in \text{set } (\text{enum-aterm } a)$

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 $\sigma \text{ a-star} = a$:

locale *annotation-problem* =

fixes *const-type* :: *string* \Rightarrow *ty*

and Γ :: *ctx*

and *a* :: *aterm*

and *a-star* :: *aterm*

and σ :: *string* \Rightarrow *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*: $\bigwedge \alpha. \alpha \in \text{tvars-ctx } \Gamma \implies \sigma \alpha = TVar \alpha$

begin

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

definition *V* :: *string set* **where**

$V = \text{tvars-aterm } a\text{-star} - \text{tvars-ctx } \Gamma$

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* \Rightarrow *string set* **where**

$\text{key } p = \{\alpha. \exists s \tau. (p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star}) \wedge \alpha \in \text{tvars-ty } \tau\} \cap V$

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

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

Positions of *a-star* and a coincide.

lemma *pos-set-eq*: $\text{pos-set } a = \text{pos-set } a\text{-star}$
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 $\exists p \in \text{pos-set } a\text{-star}. \alpha \in \text{key } p$
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 $\text{erase } a' = \text{erase } a\text{-star}$, 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 $(p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star})$.

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

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

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

lemma *consistent-with-mono*:
 $\text{consistent-with } a' P' \implies Q \subseteq P' \implies \text{consistent-with } a' Q$
unfolding *consistent-with-def* **by** *auto*

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

lemma *consistent-with-substI*:

assumes $\bigwedge p\ s\ \tau. p \in P' \implies (p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star}) \implies$
 $\text{subst-ty } \sigma' \tau = \text{subst-ty } \sigma \tau$

shows *consistent-with* (*subst-aterm* σ' *a-star*) P'

proof (*rule consistent-withI*)

fix $p\ s\ \tau$

assume $p \in P' (p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star})$

then have $\text{subst-ty } \sigma' \tau = \text{subst-ty } \sigma \tau$ **using** *assms* **by** *auto*

moreover have $(p, s, \text{subst-ty } \sigma' \tau) \in \text{set } (\text{enum-aterm } (\text{subst-aterm } \sigma' a\text{-star}))$

using $\langle (p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star}) \rangle$ *type-at-pos-subst* **by** *auto*

ultimately show $\exists \tau'. (p, s, \tau') \in \text{set } (\text{enum-aterm } (\text{subst-aterm } \sigma' a\text{-star})) \wedge$

$\tau' = \text{subst-ty } \sigma \tau$

by *auto*

qed

lemma *consistent-with-substD*:

assumes *consistent-with* (*subst-aterm* σ' *a-star*) P'

$p \in P' (p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star})$

shows $\text{subst-ty } \sigma' \tau = \text{subst-ty } \sigma \tau$

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 $P :: \text{nat set}$

assumes *P-subset*: $P \subseteq \text{pos-set } a\text{-star}$

and *coverage*: $\bigcup (\text{key } \cdot P) = V$

and *witness*: $\bigwedge p. p \in P \implies \exists \alpha \in \text{key } p. \forall p' \in P. p' \neq p \implies \alpha \notin \text{key } p'$

begin

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

lemma *coverage-mem*: $\alpha \in V \implies \exists p \in P. \alpha \in \text{key } p$

using *coverage* **by** *blast*

2.5.2 Annotations determine the substitution

lemma *sigma-agree-on-V*:

assumes *agreement*: $\bigwedge p\ s\ \tau. p \in P \implies (p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star}) \implies$
 $\text{subst-ty } \sigma' \tau = \text{subst-ty } \sigma \tau$

and $\langle \alpha \in V \rangle$

shows $\sigma' \alpha = \sigma \alpha$

proof –

obtain p **where** $p \in P \alpha \in \text{key } p$ **using** *coverage-mem* $\langle \alpha \in V \rangle$ **by** *blast*

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

The main result:

lemma $\text{annotations-determine-subst}$:

assumes $\text{agreement}: \bigwedge p \ s \ \tau. p \in P \implies (p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star}) \implies$
 $\text{subst-ty } \sigma' \tau = \text{subst-ty } \sigma \tau$
and $\text{fresh}': \bigwedge \alpha. \alpha \in \text{tvars-ctx } \Gamma \implies \sigma' \alpha = \text{TVar } \alpha$
shows $\text{subst-aterm } \sigma' a\text{-star} = a$

proof –

We show $\sigma' = \sigma$ on $\text{tvars-aterm } a\text{-star}$, then conclude by $(\bigwedge v. v \in \text{tvars-aterm } ?a \implies ?\sigma_1 v = ?\sigma_2 v) \implies \text{subst-aterm } ?\sigma_1 ?a = \text{subst-aterm } ?\sigma_2 ?a$ and $\text{subst-aterm } \sigma a\text{-star} = a$.

have $\text{agree}: \sigma' \alpha = \sigma \alpha$ **if** $\alpha \in \text{tvars-aterm } a\text{-star}$ **for** α

proof ($\text{cases } \alpha \in V$)

case True

then show $?thesis$

using $\text{sigma-agree-on-V}[\text{OF } \text{agreement}]$ **by** simp

next

case False

with that **have** $\alpha \in \text{tvars-ctx } \Gamma$ **unfolding** $V\text{-def}$ **by** auto

then have $\sigma' \alpha = \text{TVar } \alpha$ **using** fresh' **by** auto

moreover have $\sigma \alpha = \text{TVar } \alpha$ **using** $\langle \alpha \in \text{tvars-ctx } \Gamma \rangle$ freshness **by** auto

ultimately show $?thesis$ **by** simp

qed

have $\text{subst-aterm } \sigma' a\text{-star} = \text{subst-aterm } \sigma a\text{-star}$

by ($\text{intro } \text{subst-aterm-agree } \text{agree}$)

also have $\dots = a$ **using** matching **by** simp

finally show $?thesis$.

qed

2.5.3 Completeness

Theorem (Completeness). Any well-typed a' with $\text{erase } a' = \text{erase } a$ 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 $a\text{-star}$, where the substitution is identity on context variables.

theorem completeness :

assumes $a'\text{-wt}: \text{well-typed } \text{const-type } \Gamma \ a'$

and $a'\text{-erase}: \text{erase } a' = \text{erase } a$

and $\text{principality}: \exists \sigma'. \text{subst-aterm } \sigma' a\text{-star} = a'$

$\wedge (\forall \alpha \in \text{tvars-ctx } \Gamma. \sigma' \alpha = \text{TVar } \alpha)$

and *consist: consistent-with* $a' P$
shows $a' = a$
proof –
from *principality* **obtain** σ' **where**
inst: subst-aterm $\sigma' a\text{-star} = a'$ **and**
fresh': $\forall \alpha \in \text{tvars-ctx } \Gamma. \sigma' \alpha = \text{TVar } \alpha$
by *blast*
have $a' = \text{subst-aterm } \sigma' a\text{-star}$ **using** *inst* **by** *simp*
also have $\dots = a$
proof (*rule annotations-determine-subst*)
fix $p s \tau$
assume $p \in P (p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star})$
then show $\text{subst-ty } \sigma' \tau = \text{subst-ty } \sigma \tau$
using *consistent-with-substD[OF consist[folded inst] ⟨p ∈ P⟩ ⟨(p, s, τ) ∈ set (enum-aterm a-star)⟩]* **by** *auto*
next
fix α
assume $\alpha \in \text{tvars-ctx } \Gamma$
then show $\sigma' \alpha = \text{TVar } \alpha$ **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 $\text{erase } a' = \text{erase } a$ and consistent with the annotations at $P - \{p\}$.

theorem *local-minimality*:
assumes $p \in P$
shows $\exists a'. a' \neq a$
 \wedge *well-typed const-type* $\Gamma a'$
 \wedge *erase* $a' = \text{erase } a$
 \wedge *consistent-with* $a' (P - \{p\})$
proof –

Step 1: Obtain witness variable.

from *witness[OF ⟨p ∈ P⟩]* **obtain** α **where**
 α -*in-key*: $\alpha \in \text{key } p$ **and**
 α -*unique*: $\forall p' \in P. p' \neq p \longrightarrow \alpha \notin \text{key } p'$
by *blast*

Step 2: Define the altered substitution.

define σ -*star* **where** σ -*star* = $(\lambda v. \text{if } v = \alpha \text{ then } \sigma \alpha \rightarrow \sigma \alpha \text{ else } \sigma v)$

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

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

Property 1: a' differs from a .

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

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

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

Property 2: a' is well-typed.

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

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

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

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

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

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

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

```

then have  $\alpha \in \text{key } p'$ 
  using  $\langle (p', s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star}) \rangle \langle \alpha \in V \rangle$ 
  unfolding key-def by auto
  with  $\alpha\text{-unique } \langle p' \in P \rangle \langle p' \neq p \rangle$  show False by auto
qed

```

Step 2: $\sigma\text{-star} = \sigma$ on *tvars-ty* τ .

```

have  $\sigma\text{-star } \beta = \sigma \beta$  if  $\beta \in \text{tvars-ty } \tau$  for  $\beta$ 
proof –
  from that  $\langle \alpha \notin \text{tvars-ty } \tau \rangle$  have  $\beta \neq \alpha$  by auto
  then show ?thesis unfolding  $\sigma\text{-star-def}$  by simp
qed
then show subst-ty  $\sigma\text{-star } \tau = \text{subst-ty } \sigma \tau$ 
  by (rule subst-ty-agree)
qed

```

```

from  $\langle a' \neq a \rangle \langle \text{well-typed const-type } \Gamma \ a' \rangle \langle \text{erase } a' = \text{erase } a \rangle$  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  $\Rightarrow$  ty)  $\Rightarrow$  aterm  $\Rightarrow$  nat set  $\Rightarrow$  nat  $\Rightarrow$  raw-term  $\times$  nat where
  ins  $\sigma$  (AConst  $c$   $\tau$ )  $P$   $k$  =
    (if  $k \in P$  then RConstrain (RConst  $c$ ) (subst-ty  $\sigma$   $\tau$ ) else RConst  $c$ , 1)
| ins  $\sigma$  (AVar  $x$   $\tau$ )  $P$   $k$  =
  (if  $k \in P$  then RConstrain (RVar  $x$ ) (subst-ty  $\sigma$   $\tau$ ) else RVar  $x$ , 1)
| ins  $\sigma$  (AAbs  $x$   $\tau$   $a_1$   $\tau'$ )  $P$   $k$  =
  (let ( $t_1'$ ,  $n_1$ ) = ins  $\sigma$   $a_1$   $P$  ( $k + 1$ );
    t-binder = (if  $k \in P$  then RAbsT  $x$  (subst-ty  $\sigma$   $\tau$ )  $t_1'$ 
      else RAbs  $x$   $t_1'$ );
     $t'$  = (if  $k + 1 + n_1 \in P$  then RConstrain t-binder (subst-ty  $\sigma$   $\tau'$ )
      else t-binder)
    in ( $t'$ ,  $n_1 + 2$ ))
| ins  $\sigma$  (AApp  $a_1$   $a_2$   $\tau$ )  $P$   $k$  =
  (let ( $t_1'$ ,  $n_1$ ) = ins  $\sigma$   $a_1$   $P$   $k$ ;
    ( $t_2'$ ,  $n_2$ ) = ins  $\sigma$   $a_2$   $P$  ( $k + n_1$ );
    t-app = RApp  $t_1'$   $t_2'$ ;
     $t'$  = (if  $k + n_1 + n_2 \in P$  then RConstrain t-app (subst-ty  $\sigma$   $\tau$ )
      else t-app)
    in ( $t'$ ,  $n_1 + n_2 + 1$ ))

```

The top-level annotation function.

definition *annotate* :: (string \Rightarrow ty) \Rightarrow aterm \Rightarrow nat set \Rightarrow raw-term **where**
annotate σ a-star $P = \text{fst } (\text{ins } \sigma \text{ a-star } P \ 0)$

The number of positions traversed equals the node count.

lemma *ins-count*: $\text{snd } (\text{ins } \sigma \ a \ P \ k) = \text{aterm-node-count } a$
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*: $\text{strip } (\text{fst } (\text{ins } \sigma \ a \ P \ k)) = \text{erase } a$
by (*induction a arbitrary: k*) (*auto simp: Let-def case-prod-unfold*)

Corollary: annotate preserves the raw term under stripping.

corollary *strip-annotate*: $\text{strip } (\text{annotate } \sigma \ a \ P) = \text{erase } a$
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 \Rightarrow nat) \Rightarrow string set \Rightarrow bool **where**
rg-keep $\text{cnt } K = (\exists \alpha \in K. \text{cnt } \alpha \leq 1)$

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

fun *rg-fold* :: (nat \times string set) list \Rightarrow (string \Rightarrow nat) \Rightarrow nat set \times (string \Rightarrow nat)
where
rg-fold [] $\text{cnt} = (\{\}, \text{cnt})$
| *rg-fold* ((p, K) # rest) $\text{cnt} =$
 (*if* *rg-keep* $\text{cnt } K$ *then*
 $\text{let } (P', \text{cnt}') = \text{rg-fold } \text{rest } \text{cnt} \text{ in } (\text{insert } p \ P', \text{cnt}')$
 else
 $\text{let } \text{cnt}' = (\lambda \alpha. \text{if } \alpha \in K \text{ then } \text{cnt } \alpha - 1 \text{ else } \text{cnt } \alpha) \text{ in}$
 $\text{let } (P', \text{cnt}'') = \text{rg-fold } \text{rest } \text{cnt}' \text{ in } (P', \text{cnt}'')$

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*:
 $\forall (p, K) \in \text{set cands}. \alpha \notin K \implies \text{snd} (\text{rg-fold cands cnt}) \alpha = \text{cnt } \alpha$
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*: $\forall p K. (p, K) \in \text{set cands} \implies \alpha \in K \implies p \notin \text{fst} (\text{rg-fold cands cnt})$
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 $1 \leq \text{cnt } \alpha$ 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, $1 \leq \text{cnt } \alpha$ 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  $\bigwedge \alpha. \text{cnt } \alpha \geq 1$ 
  shows  $\text{snd } (\text{rg-fold cands cnt}) \alpha \geq 1$ 
  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  $\forall \alpha. (\exists (p, K) \in \text{set cands}. \alpha \in K) \longrightarrow \text{cnt1 } \alpha = \text{cnt2 } \alpha$ 
  shows  $\text{fst } (\text{rg-fold cands cnt1}) = \text{fst } (\text{rg-fold cands cnt2})$ 
  and  $\forall \alpha. (\exists (p, K) \in \text{set cands}. \alpha \in K) \longrightarrow$ 
     $\text{snd } (\text{rg-fold cands cnt1}) \alpha = \text{snd } (\text{rg-fold cands cnt2}) \alpha$ 
proof –
  have both:  $\text{fst } (\text{rg-fold cands cnt1}) = \text{fst } (\text{rg-fold cands cnt2}) \wedge$ 
     $(\forall \alpha. (\exists (p, K) \in \text{set cands}. \alpha \in K) \longrightarrow$ 
       $\text{snd } (\text{rg-fold cands cnt1}) \alpha = \text{snd } (\text{rg-fold cands cnt2}) \alpha)$ 
  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:  $\forall \alpha \in K. \text{cnt1 } \alpha = \text{cnt2 } \alpha$ 
  using Cons.prems by auto
  then have keep-eq:  $\text{rg-keep cnt1 } K = \text{rg-keep cnt2 } K$ 
  unfolding rg-keep-def by auto
  show ?case
proof (cases rg-keep cnt1 K)
  case True
  have rest-agree:  $\forall \alpha. (\exists (p, K) \in \text{set rest}. \alpha \in K) \longrightarrow \text{cnt1 } \alpha = \text{cnt2 } \alpha$ 

```

```

using Cons.premis by auto
from Cons.IH[OF rest-agree] have IH:
  fst (rg-fold rest cnt1) = fst (rg-fold rest cnt2)
   $\forall \alpha. (\exists (p, K) \in \text{set rest}. \alpha \in K) \longrightarrow$ 
    snd (rg-fold rest cnt1)  $\alpha$  = snd (rg-fold rest cnt2)  $\alpha$ 
  by auto
have snd-eq:  $\forall \beta. (\exists (p, K) \in \text{set } (pc \# \text{rest}). \beta \in K) \longrightarrow$ 
  snd (rg-fold (pc # rest) cnt1)  $\beta$  = snd (rg-fold (pc # rest) cnt2)  $\beta$ 
proof (intro allI impI)
  fix  $\beta$  assume ex:  $\exists (p, K) \in \text{set } (pc \# \text{rest}). \beta \in K$ 
  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  $\langle$ rg-keep cnt1 K $\rangle$  r1 by (simp add: Let-def)
  have fold2: rg-fold (pc # rest) cnt2 = (insert p P2, c2)
    using  $\langle$ rg-keep cnt1 K $\rangle$  keep-eq r2 by (simp add: Let-def)
  show snd (rg-fold (pc # rest) cnt1)  $\beta$  = snd (rg-fold (pc # rest) cnt2)  $\beta$ 
  proof (cases  $\exists (p, K) \in \text{set rest}. \beta \in K$ )
    case True
      from IH(2) True have c1  $\beta$  = c2  $\beta$  using r1 r2 by auto
      then show ?thesis using fold1 fold2 by simp
    next
      case False
        then have nk1:  $\forall (p, K) \in \text{set rest}. \beta \notin K$  by auto
        have snd (rg-fold rest cnt1)  $\beta$  = cnt1  $\beta$  using rg-fold-cnt-unchanged[OF
nk1] by simp
        then have c1  $\beta$  = cnt1  $\beta$  using r1 by simp
        moreover have snd (rg-fold rest cnt2)  $\beta$  = cnt2  $\beta$  using rg-fold-cnt-unchanged[OF
nk1] by simp
        then have c2  $\beta$  = cnt2  $\beta$  using r2 by simp

        moreover from ex False have  $\beta \in K$  by auto
        then have cnt1  $\beta$  = cnt2  $\beta$  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' = ( $\lambda \alpha. \text{if } \alpha \in K \text{ then cnt1 } \alpha - 1 \text{ else cnt1 } \alpha$ )
  define cnt2' where cnt2' = ( $\lambda \alpha. \text{if } \alpha \in K \text{ then cnt2 } \alpha - 1 \text{ else cnt2 } \alpha$ )
  have rest-agree:  $\forall \alpha. (\exists (p, K) \in \text{set rest}. \alpha \in K) \longrightarrow \text{cnt1}' \alpha = \text{cnt2}' \alpha$ 
  using Cons.premis 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 Falsr: 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 $\forall\alpha. (\exists(p, K) \in \text{set } ?\text{cands}. \alpha \in K) \longrightarrow ?\text{cnt1}.0 \alpha = ?\text{cnt2}.0 \alpha \implies \text{fst} (\text{rg-fold } ?\text{cands } ?\text{cnt1}.0) = \text{fst} (\text{rg-fold } ?\text{cands } ?\text{cnt2}.0)$
 $\forall\alpha. (\exists(p, K) \in \text{set } ?\text{cands}. \alpha \in K) \longrightarrow ?\text{cnt1}.0 \alpha = ?\text{cnt2}.0 \alpha \implies \forall\alpha.$

$(\exists (p, K) \in \text{set } ?\text{cands}. \alpha \in K) \longrightarrow \text{snd } (\text{rg-fold } ?\text{cands } ?\text{cnt1}.0) \alpha = \text{snd } (\text{rg-fold } ?\text{cands } ?\text{cnt2}.0) \alpha$ 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 $\forall \alpha. (\exists (p, K) \in \text{set } \text{cands}. \alpha \in K) \longrightarrow \text{cnt } \alpha \geq 1$

and $(\exists (p, K) \in \text{set } \text{cands}. \alpha \in K)$

shows $\text{snd } (\text{rg-fold } \text{cands } \text{cnt}) \alpha \geq 1$

proof –

define *cnt'* **where** $\text{cnt}' \beta = (\text{if } \exists (p, K) \in \text{set } \text{cands}. \beta \in K \text{ then } \text{cnt } \beta \text{ else } 1)$
for β

have *ge1*: $\forall \beta. \text{cnt}' \beta \geq 1$ **unfolding** *cnt'-def* **using** *assms(1)* **by** *auto*

have *agree*: $\forall \beta. (\exists (p, K) \in \text{set } \text{cands}. \beta \in K) \longrightarrow \text{cnt } \beta = \text{cnt}' \beta$

unfolding *cnt'-def* **by** *auto*

from *rg-fold-cnt-agree(2)* [*OF agree*] *assms(2)*

have $\text{snd } (\text{rg-fold } \text{cands } \text{cnt}) \alpha = \text{snd } (\text{rg-fold } \text{cands } \text{cnt}') \alpha$ **by** *auto*

then **have** $\text{snd } (\text{rg-fold } \text{cands } \text{cnt}) \alpha = \text{snd } (\text{rg-fold } \text{cands } \text{cnt}') \alpha$ **by** *simp*

also **have** $\dots \geq 1$ **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*: $\forall \alpha. \text{cnt } \alpha = \text{length } (\text{filter } (\lambda(-, K). \alpha \in K) \text{ cands}) + \text{extra } \alpha$

and *mem*: $(p, K) \in \text{set } \text{cands}$

and *kept*: $p \in \text{fst } (\text{rg-fold } \text{cands } \text{cnt})$

shows $\exists \alpha \in K. \text{extra } \alpha = 0 \wedge$

$(\forall p' K'. (p', K') \in \text{set } \text{cands} \longrightarrow p' \in \text{fst } (\text{rg-fold } \text{cands } \text{cnt}) \longrightarrow p' \neq p \longrightarrow \alpha \notin K')$

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.premis(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.premis(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.premis(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 beI[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' = ( $\lambda\alpha.$  if  $\alpha \in Kh$  then  $cnt \alpha - 1$  else  $cnt \alpha$ )
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.premis(4) <p = ph> have  $ph \in P'$  by simp
from rg-fold-subset[of rest cnt'] rec have  $P' \subseteq fst \text{ ` set rest$  by simp
with  $\langle ph \in P' \rangle$  ph-notin show ?thesis by auto
qed
next
case False

```

Target p is in the tail.

```

from Cons.premis(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.premis(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  $\alpha$ 
have cnt-rest:  $\forall \alpha. cnt \alpha = length (filter (\lambda(-, K). \alpha \in K) rest) + extra' \alpha$ 
proof
  fix  $\alpha$ 
  from Cons.premis(2) pc-def have
    cnt α = length (filter (\lambda(-, K). α ∈ K) (pc # rest)) + extra α by simp
  then show cnt α = length (filter (\lambda(-, 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  $\alpha$  where α-in: α ∈ K
  and extra'-zero: extra' α = 0
  and wit-rest:  $\forall p' K'. (p', K') \in set rest \longrightarrow$ 
    p' ∈ fst (rg-fold rest cnt)  $\longrightarrow p' \neq p \longrightarrow \alpha \notin K'$ 
  by auto

```

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

```

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

```

Head is dropped. Decrement count for variables in Kh .

```

define  $cnt'$  where  $cnt' = (\lambda \alpha. if \alpha \in Kh \text{ then } cnt \alpha - 1 \text{ else } cnt \alpha)$ 
obtain  $P'$   $cnt''$  where  $rec: rg-fold rest cnt' = (P', cnt'')$ 
  by (cases rg-fold rest cnt')
from False  $pc$ -def  $cnt'$ -def  $rec$  have  $fold$ -eq:
   $fst (rg-fold (pc \# rest) cnt) = P'$ 
  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:  $p \in fst (rg-fold rest cnt')$  by simp

```

Compute the cnt' equation for rest.

```

have  $cnt$ -rest:  $\forall \alpha. cnt' \alpha = length (filter (\lambda(-, K). \alpha \in K) rest) + extra \alpha$ 
proof
  fix  $\alpha$ 
  from Cons.prems(2)  $pc$ -def have
     $cnt$ -full:  $cnt \alpha = (if \alpha \in Kh \text{ then } 1 \text{ else } 0) + length (filter (\lambda(-, K). \alpha \in$ 
     $K) rest) + extra \alpha$ 
  by simp
  show  $cnt' \alpha = length (filter (\lambda(-, K). \alpha \in K) rest) + extra \alpha$ 
  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 False: 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  $\alpha$  where  $\alpha$ -in:  $\alpha \in K$ 
and extra-zero: extra  $\alpha = 0$ 
and wit-rest:  $\forall p' K'. (p', K') \in \text{set rest} \longrightarrow$ 
   $p' \in \text{fst (rg-fold rest cnt')} \longrightarrow p' \neq p \longrightarrow \alpha \notin K'$ 
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  $P' \subseteq \text{fst ' set rest}$  by simp
with ph-notin have ph-notin-P:  $ph \notin P'$  by auto
have wit-full:  $\forall p' K'. (p', K') \in \text{set (pc \# rest)} \longrightarrow$ 
   $p' \in \text{fst (rg-fold (pc \# rest) cnt')} \longrightarrow p' \neq p \longrightarrow \alpha \notin K'$ 
proof (intro allI impI)
  fix  $p' K'$ 
  assume  $p'$ -in:  $(p', K') \in \text{set (pc \# rest)}$ 
  and  $p'$ -kept:  $p' \in \text{fst (rg-fold (pc \# rest) cnt')}$ 
  and  $p'$ -ne:  $p' \neq p$ 
  from  $p'$ -kept fold-eq have  $p' \in P'$  by simp
  with ph-notin-P have  $p' \neq ph$  by auto
  from  $p'$ -in pc-def  $\langle p' \neq ph \rangle$  have  $(p', K') \in \text{set rest}$  by auto
  from  $\langle p' \in P' \rangle$  rec have  $p' \in \text{fst (rg-fold rest cnt')}$  by simp
  from wit-rest  $\langle (p', K') \in \text{set rest} \rangle$  this  $p'$ -ne
  show  $\alpha \notin K'$  by auto

```

qed

```

from  $\alpha$ -in extra-zero wit-full show ?thesis by (intro becI[of -  $\alpha$ ]) auto

```

qed

qed

qed

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

lemma rg-fold-witness:

```

assumes distinct (map fst cands)
and  $\forall \alpha. \text{cnt } \alpha = \text{length (filter (\lambda(-, K). \alpha \in K) cands)}$ 
and  $(p, K) \in \text{set cands}$  and  $p \in \text{fst (rg-fold cands cnt)}$ 
shows  $\exists \alpha \in K. \forall p' \in \text{fst (rg-fold cands cnt)}. p' \neq p \longrightarrow$ 
   $(\forall K'. (p', K') \in \text{set cands} \longrightarrow \alpha \notin K')$ 
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* ($\lambda(p, s, \tau). (p, \text{tvars-ty } \tau \cap V)$) (*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 $\alpha \in V$
shows *init-count candidates* $\alpha \geq 1$
proof –
from *coverage-initial[OF assms]* **obtain** *p* **where**
 $p \in \text{pos-set } a\text{-star}$ $\alpha \in \text{key } p$ **by** *auto*
from $\langle \alpha \in \text{key } p \rangle$ **obtain** *s* τ **where**
 $\text{mem}: (p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star})$ **and** $\text{tv}: \alpha \in \text{tvars-ty } \tau$ **and** $\alpha \in V$
unfolding *key-def* **by** *auto*
then have *mem-cand*: $(p, \text{tvars-ty } \tau \cap V) \in \text{set } \text{candidates}$
unfolding *candidates-def* **by** *force*
have $\alpha \in \text{tvars-ty } \tau \cap V$ **using** *tv assms* **by** *auto*
with *mem-cand* **have** $(p, \text{tvars-ty } \tau \cap V) \in \text{set } (\text{filter } (\lambda(-, K). \alpha \in K) \text{ candidates})$
by *auto*
then have $0 < \text{length } (\text{filter } (\lambda(-, K). \alpha \in K) \text{ candidates})$
by (*rule length-pos-if-in-set*)
then have $\text{length } (\text{filter } (\lambda(-, K). \alpha \in K) \text{ candidates}) \geq 1$ **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* \subseteq *pos-set a-star*
proof –
have *reverse-greedy candidates* \subseteq *fst ' set candidates*
unfolding *reverse-greedy-def* **using** *rg-fold-subset* **by** *auto*
also have $\dots \subseteq \text{pos-set } a\text{-star}$
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 $(p, K) \in \text{set candidates}$
shows $K = \text{key } p$
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*: $\bigcup (\text{key } \text{reverse-greedy candidates}) = V$
proof (*intro equalityI subsetI*)
fix α **assume** $\alpha \in \bigcup (\text{key } \text{reverse-greedy candidates})$
then obtain p **where** $p \in \text{reverse-greedy candidates } \alpha \in \text{key } p$ **by** *auto*
then show $\alpha \in V$ **unfolding** *key-def* **by** *auto*
next
fix α **assume** $\alpha \in V$

By $\llbracket \forall \alpha. (\exists (p, K) \in \text{set } ?\text{cands}. \alpha \in K) \longrightarrow 1 \leq ?\text{cnt } \alpha; \exists (p, K) \in \text{set } ?\text{cands}. ?\alpha \in K \rrbracket \implies 1 \leq \text{snd } (\text{rg-fold } ?\text{cands } ?\text{cnt}) ?\alpha$, the final count for α is ≥ 1 . This means at least one candidate covering α was not dropped.

from *coverage-initial*[*OF* $\alpha \in V$] **obtain** $p0$ **where**
 $p0 \in \text{pos-set a-star } \alpha \in \text{key } p0$ **by** *auto*
from $\langle \alpha \in \text{key } p0 \rangle$ **obtain** $s0 \tau0$ **where**
 $\text{mem0}: (p0, s0, \tau0) \in \text{set } (\text{enum-aterm a-star})$ **and** $\alpha \in \text{tvars-ty } \tau0 \alpha \in V$
unfolding *key-def* **by** *auto*
then have $(p0, \text{tvars-ty } \tau0 \cap V) \in \text{set candidates}$
unfolding *candidates-def* **by** *force*
then have *ex-cand*: $\exists (p, K) \in \text{set candidates}. \alpha \in K$
using $\langle \alpha \in \text{tvars-ty } \tau0 \rangle \langle \alpha \in V \rangle$ **by** (*intro bexI*[*of* - $(p0, \text{tvars-ty } \tau0 \cap V)$])
auto
have *cnt-ge1*: $\forall \beta. (\exists (p, K) \in \text{set candidates}. \beta \in K) \longrightarrow \text{init-count candidates } \beta \geq 1$
proof (*intro allI impI*)
fix β **assume** $\exists (p, K) \in \text{set candidates}. \beta \in K$
then obtain $p1 K1$ **where** $(p1, K1) \in \text{set candidates } \beta \in K1$ **by** *auto*
from *candidates-key-eq*[*OF* *this*(1)] $\langle \beta \in K1 \rangle$ **have** $\beta \in \text{key } p1$ **by** *auto*
then have $\beta \in V$ **unfolding** *key-def* **by** *auto*
then show $\text{init-count candidates } \beta \geq 1$ **using** *candidates-count-ge1* **by** *auto*
qed
from *rg-fold-preserves-ge1-on-keys*[*OF* *cnt-ge1 ex-cand*]
have *final-ge1*: $\text{snd } (\text{rg-fold candidates } (\text{init-count candidates})) \alpha \geq 1$.

Final count ≥ 1 means at least one position covering α was kept. Since final count ≥ 1 , at least one candidate with α was kept.

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

```

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

```

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

```

have nk:  $\forall p K. (p, K) \in \text{set candidates} \longrightarrow \alpha \in K \longrightarrow p \notin \text{fst (rg-fold candidates (init-count candidates))}$ 
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:  $(pk, Kk) \in \text{set candidates}$ 
and  $\alpha \in Kk$  and pk-kept:  $pk \in \text{fst (rg-fold candidates (init-count candidates))}$ 
by auto
from pk-kept have  $pk \in \text{reverse-greedy candidates}$  unfolding reverse-greedy-def
by simp
from candidates-key-eq[OF pk-mem]  $\langle \alpha \in Kk \rangle$  have  $\alpha \in \text{key } pk$  by auto
from  $\langle pk \in \text{reverse-greedy candidates} \rangle \langle \alpha \in \text{key } pk \rangle$ 
show  $\alpha \in \bigcup (\text{key } \text{'reverse-greedy candidates'})$  by auto
qed

```

Witness property: each kept position has a witness variable.

lemma *rg-witness*:

```

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

from rg-fold-witness[OF candidates-distinct - pK]
have wit:  $\exists \alpha \in K. \forall p' \in \text{fst (rg-fold candidates (init-count candidates))}. p' \neq p$ 
 $\longrightarrow$ 
 $(\forall K'. (p', K') \in \text{set candidates} \longrightarrow \alpha \notin K')$ 
by (simp add: init-count-def)
then obtain  $\alpha$  where  $\alpha \in K$  and
 $\text{excl: } \forall p' \in \text{fst (rg-fold candidates (init-count candidates))}. p' \neq p \longrightarrow$ 
 $(\forall K'. (p', K') \in \text{set candidates} \longrightarrow \alpha \notin K')$  by auto
from candidates-key-eq[OF pK1]  $\langle \alpha \in K \rangle$  have  $\alpha \in \text{key } p$  by simp

```

moreover have $\forall p' \in \text{reverse-greedy candidates. } p' \neq p \longrightarrow \alpha \notin \text{key } p'$
proof (*intro ballI impI*)
fix p' **assume** $p' \in \text{reverse-greedy candidates } p' \neq p$
then have $p'\text{-in: } p' \in \text{fst (rg-fold candidates (init-count candidates))}$
unfolding *reverse-greedy-def* **by** *simp*
from *rg-fold-subset* **have** $p' \in \text{fst ' set candidates}$
using $p'\text{-in}$ **by** (*metis in-mono*)
then obtain K' **where** $(p', K') \in \text{set candidates}$ **by** *auto*
from *excl p'-in <p' ≠ p>* **this have** $\alpha \notin K'$ **by** *auto*
from *candidates-key-eq[OF <(p', K') ∈ set candidates>]* **this**
show $\alpha \notin \text{key } p'$ **by** *simp*
qed
ultimately show *?thesis* **by** *auto*
qed

Instantiate *annotation-selection* with $P = \text{reverse-greedy candidates}$. 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* \subseteq *pos-set a-star*
by (*rule rg-result-subset*)
show $\bigcup (\text{key ' reverse-greedy candidates}) = V$
by (*rule rg-coverage*)
fix p **assume** $p \in \text{reverse-greedy candidates}$
then show $\exists \alpha \in \text{key } p. \forall p' \in \text{reverse-greedy candidates. } p' \neq p \longrightarrow \alpha \notin \text{key } p'$
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**
 $t\text{-out} = \text{annotate } \sigma \text{ a-star (reverse-greedy candidates)}$

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 $\exists \sigma'. \text{subst-aterm } \sigma' \text{ a-star} = a' \wedge (\forall \alpha \in \text{tvars-ctx } \Gamma. \sigma' \alpha = \text{TVar } \alpha)$
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 $E :: 'a \text{ set}$ **and** $indep :: 'a \text{ set} \Rightarrow \text{bool}$

assumes *finite-E*: $\text{finite } E$

assumes *empty-indep*: $indep \{\}$

assumes *indep-subset-carrier*: $\bigwedge F. indep\ F \Longrightarrow F \subseteq E$

assumes *hereditary*: $\bigwedge F\ F'. indep\ F \Longrightarrow F' \subseteq F \Longrightarrow indep\ F'$

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

definition (**in** *indep-system*) $maximal :: 'a \text{ set} \Rightarrow \text{bool}$ **where**
 $maximal\ F \longleftrightarrow indep\ F \wedge (\forall e \in E - F. \neg indep\ (F \cup \{e\}))$

lemma (**in** *indep-system*) $maximalI$:

assumes $indep\ F \wedge e. e \in E - F \Longrightarrow \neg indep\ (F \cup \{e\})$

shows $maximal\ F$

unfolding *maximal-def* **using** *assms* **by** *blast*

lemma (**in** *indep-system*) $maximalD$:

assumes $maximal\ F$

shows $indep\ F \wedge e. e \in E - F \Longrightarrow \neg indep\ (F \cup \{e\})$

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 $greedy-step :: ('a \text{ set} \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$ **where**

$greedy\text{-}step\ indep\ e\ F = (if\ indep\ (F \cup \{e\})\ then\ F \cup \{e\}\ else\ F)$

definition $best\text{-}in\text{-}greedy :: ('a\ set \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ set\ \mathbf{where}$
 $best\text{-}in\text{-}greedy\ indep\ es = foldl\ (\lambda F\ e.\ greedy\text{-}step\ indep\ e\ F)\ \{\}\ es$

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

fun $greedy\text{-}partial :: ('a\ set \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'a\ set\ \mathbf{where}$
 $greedy\text{-}partial\ indep\ es\ 0 = \{\}$ |
 $greedy\text{-}partial\ indep\ es\ (Suc\ i) =$
 $greedy\text{-}step\ indep\ (es\ !\ i)\ (greedy\text{-}partial\ indep\ es\ i)$

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

lemma $best\text{-}in\text{-}greedy\text{-}foldl\text{-}partial$:

assumes $n \leq length\ es$

shows $foldl\ (\lambda F\ e.\ greedy\text{-}step\ indep\ e\ F)\ \{\}\ (take\ n\ es) = greedy\text{-}partial\ indep\ es\ n$

using $assms$

proof ($induction\ n$)

case 0

then show $?case$ **by** $simp$

next

case ($Suc\ n$)

thus $?case$

by ($simp\ add: take\text{-}Suc\text{-}conv\text{-}app\text{-}nth$)

qed

lemma $best\text{-}in\text{-}greedy\text{-}eq\text{-}partial$:

$best\text{-}in\text{-}greedy\ indep\ es = greedy\text{-}partial\ indep\ es\ (length\ es)$

unfolding $best\text{-}in\text{-}greedy\text{-}def$

using $best\text{-}in\text{-}greedy\text{-}foldl\text{-}partial[of\ length\ es\ es\ indep]$

by $simp$

3.2.1 Properties of $greedy\text{-}partial$

The partial result is monotonically increasing.

lemma $greedy\text{-}partial\text{-}mono$:

$i \leq j \implies greedy\text{-}partial\ indep\ es\ i \subseteq greedy\text{-}partial\ indep\ es\ j$

proof ($induction\ j$)

case 0

then show $?case$ **by** $simp$

next

case ($Suc\ j$)

then show $?case$

by ($cases\ i = Suc\ j$) ($auto\ simp: greedy\text{-}step\text{-}def$)

qed

If an element was skipped at step j , adding it to F_{j-1} was not independent.

lemma *greedy-partial-skip*:
assumes $j < \text{length } es$ $es ! j \notin \text{greedy-partial indep } es$ (*Suc j*)
shows $\neg \text{indep } (\text{greedy-partial indep } es j \cup \{es ! j\})$
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 $i \leq \text{length } es$
shows $\text{greedy-partial indep } es i \subseteq \text{set } (\text{take } i \text{ } es)$
using *assms*
proof (*induction i*)
case 0
then show *?case* **by** *simp*
next
case (*Suc i*)
then have $\text{greedy-partial indep } es i \subseteq \text{set } (\text{take } i \text{ } es)$ **by** *simp*
moreover have $\text{set } (\text{take } i \text{ } es) \subseteq \text{set } (\text{take } (\text{Suc } i) \text{ } es)$
by (*simp add: set-take-subset-set-take*)
moreover have $es ! i \in \text{set } (\text{take } (\text{Suc } i) \text{ } es)$
using *Suc.prem*s **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 $i \leq \text{length } es$ $\text{set } es \subseteq E$
shows $\text{indep } (\text{greedy-partial indep } es i)$
using *assms*
proof (*induction i*)
case 0
then show *?case* **using** *empty-indep* **by** *simp*
next
case (*Suc i*)
then have *IH*: $\text{indep } (\text{greedy-partial indep } es i)$ **by** *simp*
show *?case*
proof (*cases indep (greedy-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  $\subseteq$  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  $\in$  E – best-in-greedy indep es
  show  $\neg$  indep (best-in-greedy indep es  $\cup$  {e})
  proof –
    from e-outside perm have e  $\in$  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  $\notin$  greedy-partial indep es (length es)
      unfolding best-in-greedy-eq-partial by simp
    then have e  $\notin$  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:  $\neg$  indep (greedy-partial indep es j  $\cup$  {e})
      using greedy-partial-skip[of j es indep] j by simp
    have sub: greedy-partial indep es j  $\cup$  {e}  $\subseteq$  best-in-greedy indep es  $\cup$  {e}
      unfolding best-in-greedy-eq-partial
      using greedy-partial-mono[of j length es indep es] j by auto
    show  $\neg$  indep (best-in-greedy indep es  $\cup$  {e})
    proof
      assume indep (best-in-greedy indep es  $\cup$  {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  $S :: \text{nat set}$ 
    and  $V :: \text{string set}$ 
    and  $\text{key} :: \text{nat} \Rightarrow \text{string set}$ 
  assumes finite-S:  $\text{finite } S$ 
  assumes finite-V:  $\text{finite } V$ 
  assumes key-subset:  $\bigwedge p. p \in S \implies \text{key } p \subseteq V$ 
  assumes S-covers-V:  $\bigcup (\text{key } ` S) = V$ 
begin

```

Coverage

```

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

```

```

lemma covers-S:  $\text{covers } S$ 
  unfolding covers-def using S-covers-V by blast

```

```

lemma covers-mono:
  assumes  $\text{covers } Q \ Q \subseteq Q'$ 
  shows  $\text{covers } Q'$ 
  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: $F = \{F \subseteq S \mid S \setminus F \text{ covers } V\}$. 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 ::  $\text{nat set} \Rightarrow \text{bool}$  where
  ann-indep  $F \longleftrightarrow F \subseteq S \wedge \text{covers } (S - F)$ 

```

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 $F \implies F \subseteq S$
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** $\langle F \subseteq S \rangle$ **by** *blast*
moreover **have** $S - F \subseteq S - F'$ **using** $\langle F' \subseteq F \rangle$ **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 $\bigwedge F. \text{ann-indep } F \implies F \subseteq S$ **by** (*rule ann-indep-subset-carrier*)
show $\bigwedge F \ F'. \text{ann-indep } F \implies F' \subseteq F \implies \text{ann-indep } F'$
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 $1 < \text{var-count } F \ \alpha$ for every $\alpha \in \text{key } p$, where *var-count* α counts positions in $S - F$ covering α .

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

lemma *feasibility-iff-count*:

assumes *ann-indep* $F \ p \in S - F$
shows *ann-indep* ($F \cup \{p\}$) $\iff (\forall \alpha \in \text{key } p. \text{var-count } F \ \alpha > 1)$

proof

assume *indep-Fp*: *ann-indep* ($F \cup \{p\}$)
show $\forall \alpha \in \text{key } p. \text{var-count } F \ \alpha > 1$

proof

fix α **assume** α -*in*: $\alpha \in \text{key } p$
from *indep-Fp* **have** *covers* ($S - (F \cup \{p\})$) **unfolding** *ann-indep-def* **by** *simp*

then have $V\text{-cov}$: $V \subseteq \bigcup(\text{key } '(S - (F \cup \{p\})))$ **unfolding** *covers-def* .
from $\alpha\text{-in } \text{assms}(2)$ *key-subset* **have** $\alpha \in V$ **by** *blast*
then obtain q **where** $q: q \in S - (F \cup \{p\}) \ \alpha \in \text{key } q$
using $V\text{-cov}$ **by** *blast*
then have $q \neq p \ q \in S - F$ **by** *auto*
then have $\{p, q\} \subseteq \{r \in S - F. \ \alpha \in \text{key } r\}$
using $\alpha\text{-in } q(2)$ *assms(2)* **by** *blast*
moreover have $p \neq q$ **using** $\langle q \neq p \rangle$ **by** *simp*
moreover have *finite* $\{r \in S - F. \ \alpha \in \text{key } r\}$ **using** *finite-S* **by** *auto*
ultimately have *card* $\{r \in S - F. \ \alpha \in \text{key } r\} \geq 2$
by (*metis card-2-iff card-mono*)
then show *var-count* $F \ \alpha > 1$ **unfolding** *var-count-def* **by** *simp*
qed
next
assume *count-cond*: $\forall \alpha \in \text{key } p. \ \text{var-count } F \ \alpha > 1$
have $F \cup \{p\} \subseteq S$ **using** *assms* **unfolding** *ann-indep-def* **by** *auto*
moreover have *covers* $(S - (F \cup \{p\}))$
proof –
have $V\text{-sub}$: $V \subseteq \bigcup(\text{key } '(S - F))$
using *assms(1)* **unfolding** *ann-indep-def* *covers-def* **by** *simp*
show *?thesis* **unfolding** *covers-def*
proof
fix α **assume** $\alpha \in V$
then obtain q **where** $q: q \in S - F \ \alpha \in \text{key } q$
using $V\text{-sub}$ **by** *blast*
show $\alpha \in \bigcup(\text{key } '(S - (F \cup \{p\})))$
proof (*cases* $\alpha \in \text{key } p$)
case *True*
then have *cnt-gt*: *card* $\{r \in S - F. \ \alpha \in \text{key } r\} > 1$
using *count-cond* **unfolding** *var-count-def* **by** *simp*
moreover have $p\text{-mem}$: $p \in \{r \in S - F. \ \alpha \in \text{key } r\}$ **using** *True* *assms(2)*
by *simp*
moreover have *fin*: *finite* $\{r \in S - F. \ \alpha \in \text{key } r\}$ **using** *finite-S* **by** *auto*
ultimately have *card* $(\{r \in S - F. \ \alpha \in \text{key } r\} - \{p\}) > 0$
by (*simp add: card-Diff-singleton*)
then have $\{r \in S - F. \ \alpha \in \text{key } r\} - \{p\} \neq \{\}$
using *card-gt-0-iff fin finite-Diff* **by** *blast*
then obtain r **where** $r \in S - F \ \alpha \in \text{key } r \ r \neq p$ **by** *blast*
then show *?thesis* **by** *blast*
next
case *False*
then have $q \neq p$ **using** $q(2)$ **by** *blast*
then have $q \in S - (F \cup \{p\})$ **using** $q(1)$ **by** *blast*
then show *?thesis* **using** $q(2)$ **by** *blast*
qed
qed
qed
ultimately show *ann-indep* $(F \cup \{p\})$ **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 ∈ S - F-star*

shows \neg *covers (S - F-star - {p})*

proof -

from *greedy-maximal p-in-P* **have** \neg *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** \neg *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 ∈ S - F-star*

obtains α **where** $\alpha \in \text{key } p \ \alpha \in V$

$\bigwedge q. q \in S - F-star - \{p\} \implies \alpha \notin \text{key } q$

proof -

let *?P = S - F-star*

have *cov: covers ?P*

using *maximalD(1)[OF greedy-maximal] local-minimality-coverage by simp*

have *not-cov: \neg covers (?P - {p})*

using *local-minimality[OF greedy-maximal p-in-P] by simp*

from *not-cov* **obtain** α **where** $\alpha \in V \ \alpha \notin \bigcup (\text{key } ' (?P - \{p\}))$

unfolding *covers-def by blast*

from *cov* **have** $V \subseteq \bigcup (\text{key } ' ?P)$ **unfolding** *covers-def by simp*

with $\alpha(1)$ **obtain** q **where** $q \in ?P \ \alpha \in \text{key } q$ **by** *blast*

with $\alpha(2)$ **have** $q = p$ **by** *blast*

then **have** $\alpha \in \text{key } p$ **using** $\langle \alpha \in \text{key } q \rangle$ **by** *simp*

moreover **have** $\bigwedge q. q \in ?P - \{p\} \implies \alpha \notin \text{key } q$

using $\alpha(2)$ **by** *blast*

ultimately **show** *?thesis using that $\alpha(1)$ by blast*

qed

3.6 Reverse Greedy Correctness

definition $\langle \text{reverse-greedy}' \text{ es} = S - \text{best-in-greedy ann-indep es} \rangle$

theorem *annotation-algorithm-correct*:

assumes *perm: set es = S* **and** *distinct: distinct es*

```

shows covers (reverse-greedy' es)
  and  $\bigwedge p. p \in \text{reverse-greedy}' \text{ es} \implies$ 
     $\exists \alpha \in \text{key } p. \alpha \in V \wedge (\forall q \in \text{reverse-greedy}' \text{ es} - \{p\}. \alpha \notin \text{key } q)$ 
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  $\bigwedge p. p \in \text{reverse-greedy}' \text{ es} \implies$ 
     $\exists \alpha \in \text{key } p. \alpha \in V \wedge (\forall q \in \text{reverse-greedy}' \text{ es} - \{p\}. \alpha \notin \text{key } q)$ 
    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  $\bigwedge p. p \in \text{pos-set } a\text{-star} \implies \text{key } p \subseteq V$ 
    unfolding key-def by auto
  show  $\bigcup (\text{key } \text{' pos-set } a\text{-star}) = V$ 
  proof
    show  $\bigcup (\text{key } \text{' pos-set } a\text{-star}) \subseteq V$  unfolding key-def by auto
  next
    show  $V \subseteq \bigcup (\text{key } \text{' pos-set } a\text{-star})$ 
  proof
    fix  $\alpha$  assume  $\alpha \in V$ 
    then have  $\alpha \in \text{tvars-aterm } a\text{-star}$  unfolding V-def by auto
    then obtain  $p \ s \ \tau$  where  $(p, s, \tau) \in \text{set } (\text{enum-aterm } a\text{-star})$   $\alpha \in \text{tvars-ty } \tau$ 
      using tvars-aterm-subset-enum by blast
    then have  $p \in \text{pos-set } a\text{-star}$  unfolding pos-set-def by force
    moreover have  $\alpha \in \text{key } p$  unfolding key-def using  $\langle (p, s, \tau) \in \cdot \rangle \langle \alpha \in \text{tvars-ty } \tau \rangle \langle \alpha \in V \rangle$  by auto
    ultimately show  $\alpha \in \bigcup (\text{key } \text{' pos-set } a\text{-star})$  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:  $\langle \text{ann.reverse-greedy}' \text{ es-from-enum} \subseteq \text{pos-set } a\text{-star} \rangle$ 
by (simp add: ann.reverse-greedy'-def)

lemma reverse-greedy'-coverage:  $\langle \bigcup (\text{key } \text{' ann.reverse-greedy}' \text{ es-from-enum}) = V \rangle$ 
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  $\langle p \in \text{ann.reverse-greedy}' \text{ es-from-enum} \rangle$ 
shows  $\langle \exists \alpha \in \text{key } p. \forall p' \in \text{ann.reverse-greedy}' \text{ es-from-enum}. p' \neq p \longrightarrow \alpha \notin \text{key } p' \rangle$ 
using assms ann.annotation-algorithm-correct(2)[OF es-set es-distinct, of p]
by auto

interpretation ann: annotation-selection
  const-type
   $\Gamma a \text{ } a\text{-star } \sigma \langle \text{ann.reverse-greedy}' \text{ es-from-enum} \rangle$ 
using reverse-greedy'-subs-pos reverse-greedy'-coverage reverse-greedy'-witness
by unfold-locales auto

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

```