Type Annotations with Roundtrip Property

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

June 4, 2026

Abstract

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

Contents

theory *Smolka*
  imports *Main*
begin

1 Human-authored proof formalization

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

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

1.1 Types

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

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

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

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

1.1.1 Type variables of a type

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

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

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

lemma *tvars-mty-None*[*simp*]: *tvars-mty None* = {}
  ⟨*proof*⟩

lemma *tvars-mty-Some*[*simp*]: *tvars-mty* (*Some* σ) = *tvars-ty* σ
  ⟨*proof*⟩

lemma *tvars-ty-finite*[*simp*, *intro*]: *finite* (*tvars-ty* τ)
⟨*proof*⟩

lemma *tvars-mty-finite*[*simp*, *intro*]: *finite* (*tvars-mty* ξ)
⟨*proof*⟩

1.1.2 Type substitution

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

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

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

lemma *subst-mty-None*[*simp*]: *subst-mty* ϱ *None* = *None*
⟨*proof*⟩

lemma *subst-mty-Some*[*simp*]: *subst-mty* ϱ (*Some* σ) = *Some* (*subst-ty* ϱ σ)
⟨*proof*⟩

1.1.3 Substitution composition and single substitution

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

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

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

lemma *single-subst-same*[*simp*]: *single-subst* τ α α = τ
⟨*proof*⟩

lemma *single-subst-other*[*simp*]: *a* ≠ α ⟹ *single-subst* τ α *a* = *TyVar a*
⟨*proof*⟩

1.1.4 Basic properties of substitution

lemma *subst-ty-comp*:
subst-ty ϱ (*subst-ty* ϱ' τ) = *subst-ty* (ϱ ∘∘ ϱ') τ
⟨*proof*⟩

lemma *subst-ty-id*[*simp*]: *subst-ty TyVar* $\tau = \tau$
⟨*proof*⟩

lemma *subst-mty-id*[*simp*]: *subst-mty TyVar* $\xi = \xi$
⟨*proof*⟩

lemma *subst-mty-comp*: *subst-mty* ϱ (*subst-mty* ϱ' ξ) = *subst-mty* (ϱ ∘∘ ϱ') ξ
⟨*proof*⟩

lemma *subst-ty-agree*:
subst-ty ϱ τ = *subst-ty* ϱ' τ ⟷ (∀ α ∈ *tvars-ty* τ. ϱ α = ϱ' α)
⟨*proof*⟩

lemma *subst-ty-cong*:
(⋀α. α ∈ *tvars-ty* τ ⟹ ϱ α = ϱ' α) ⟹ *subst-ty* ϱ τ = *subst-ty* ϱ' τ
⟨*proof*⟩

lemma *subst-ty-swap*:
assumes $\beta \notin$ *tvars-ty* τ
shows *subst-ty* (*single-subst* (*TyVar* α) β) (*subst-ty* (*single-subst* (*TyVar* β) α) τ) = τ
⟨*proof*⟩

Substitution distributes over *tvars-ty*.

lemma *tvars-ty-subst*: *tvars-ty* (*subst-ty* ϱ τ) = ⋃(*tvars-ty* ‘ ϱ ‘ *tvars-ty* τ)
⟨*proof*⟩

lemma *tvars-mty-subst*:
tvars-mty (*subst-mty* ϱ ξ) = ⋃(*tvars-ty* ‘ ϱ ‘ *tvars-mty* ξ)
⟨*proof*⟩

1.2 Terms

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

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

1.2.1 Type variables of a term

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

fun *tvars-trm* :: (*'tv*,*'k*,*'v*,*'c*) *trm* ⇒ *'tv set* **where**
tvars-trm (*Vr x* ξ) = *tvars-mty* ξ
| *tvars-trm* (*Ct c* ξ) = *tvars-mty* ξ
| *tvars-trm* (*App t1 t2* ξ) = *tvars-trm t1* ∪ *tvars-trm t2* ∪ *tvars-mty* ξ

| *tvars-trm* (*Lam x ξ t ζ*) = *tvars-mty ξ* ∪ *tvars-trm t* ∪ *tvars-mty ζ*

lemma *tvars-trm-finite*[*simp*, *intro*]: *finite* (*tvars-trm t*)
⟨*proof*⟩

1.2.2 Substitution on terms

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

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

lemma *subst-trm-comp*:
subst-trm ϱ (*subst-trm ϱ'* *t*) = *subst-trm* (*ϱ* ∘∘ *ϱ'*) *t*
⟨*proof*⟩

lemma *subst-trm-id*[*simp*]: *subst-trm TyVar t* = *t*
⟨*proof*⟩

lemma *subst-trm-cong*:
(⋀α. α ∈ *tvars-trm t* ⟹ *ϱ* α = *ϱ'* α) ⟹ *subst-trm ϱ t* = *subst-trm ϱ'* *t*
⟨*proof*⟩

lemma *subst-trm-agree*:
subst-trm ϱ t = *subst-trm ϱ'* *t* ⟷ (∀ α ∈ *tvars-trm t*. *ϱ* α = *ϱ'* α)
⟨*proof*⟩

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

lemma *subst-trm-id-on-tvars*:
assumes *subst-trm ϱ v* = *v* α ∈ *tvars-trm v*
shows *ϱ* α = *TyVar* α
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *tvars-trm-subst*:
tvars-trm (*subst-trm ϱ t*) = ⋃(*tvars-ty* ‘ *ϱ* ‘ *tvars-trm t*)
⟨*proof*⟩

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

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

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

lemma *fterm-subst*[*simp*]: *fterm v* $\Longrightarrow$ *fterm* (*subst-trm* ϱ *v*)
  ⟨*proof*⟩

1.2.4 Unambiguity

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

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

lemma *subst-trm-bvars*[*simp*]: *bvars* (*subst-trm* ϱ *t*) = *bvars t*
  ⟨*proof*⟩

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

lemma *subst-trm-unambiguous*[*simp*]: *unambiguous* (*subst-trm* ϱ *t*) = *unambiguous t*
⟨*proof*⟩

1.3 Positions

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

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

lemma *poss-finite*[*simp*, *intro*]: *finite* (*poss t*)
⟨*proof*⟩

lemma *nil-in-poss*[*simp*]: [] ∈ *poss t*
⟨*proof*⟩

lemma *subst-trm-poss*[*simp*]: *poss* (*subst-trm* ϱ *t*) = *poss t*
⟨*proof*⟩

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

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

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

lemma *mtp-of-subst*:
assumes *p* ∈ *poss t*
shows *mtp-of* (*subst-trm* ϱ *t*) *p* = *subst-mty* ϱ (*mtp-of t p*)
⟨*proof*⟩

lemma *tvars-trm-eq-UN-mtp*:

tvars-trm t = (⋃ *p* ∈ *poss t*. *tvars-mty* (*mtp-of t p*))
⟨*proof*⟩

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *remove-annot-poss*:
assumes *p* ∈ *poss t*
shows *poss* (*remove-annot t p*) = *poss t*
⟨*proof*⟩

lemma *remove-annot-bvars*:
assumes *p* ∈ *poss t*
shows *bvars* (*remove-annot t p*) = *bvars t*
⟨*proof*⟩

lemma *remove-annot-unambiguous*:
assumes *p* ∈ *poss t unambiguous t*
shows *unambiguous* (*remove-annot t p*)
⟨*proof*⟩

Erase: remove all type annotations.

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

lemma *erase-uterm*[*simp*]: *uterm* (*erase t*)
  ⟨*proof*⟩

lemma *erase-poss*[*simp*]: *poss* (*erase t*) = *poss t*
  ⟨*proof*⟩

lemma *erase-bvars*[*simp*]: *bvars* (*erase t*) = *bvars t*
  ⟨*proof*⟩

lemma *erase-unambiguous*[*simp*]: *unambiguous* (*erase t*) = *unambiguous t*
  ⟨*proof*⟩

1.4 Completion relations for types and terms

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

lemma *mcompl-None*[*simp*, *intro*]: *None* $\sqsubseteq_m$ ζ
  ⟨*proof*⟩

lemma *mcompl-refl*[*simp*, *intro*]: $\xi \sqsubseteq_m \xi$
  ⟨*proof*⟩

lemma *mcompl-Some-iff*[*simp*]: *Some* $\sigma \sqsubseteq_m \zeta \longleftrightarrow \zeta$ = *Some* σ
  ⟨*proof*⟩

lemma *mcompl-antisym*: $[\![\xi \sqsubseteq_m \zeta; \zeta \sqsubseteq_m \xi]\!] \Longrightarrow \xi = \zeta$
  ⟨*proof*⟩

lemma *mcompl-trans*: $[\![\xi \sqsubseteq_m \zeta; \zeta \sqsubseteq_m \chi]\!] \Longrightarrow \xi \sqsubseteq_m \chi$
  ⟨*proof*⟩

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

1.4.1 Inversion rules

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

lemma *compl-Vr-leftE*:

assumes $Vr\ x\ \xi \sqsubseteq t$
obtains ξ' **where** $\xi \sqsubseteq_m \xi'\ t = Vr\ x\ \xi'$
⟨*proof*⟩

lemma *compl-Ct-leftE*:
assumes $Ct\ c\ \xi \sqsubseteq t$
obtains ξ' **where** $\xi \sqsubseteq_m \xi'\ t = Ct\ c\ \xi'$
⟨*proof*⟩

lemma *compl-App-leftE*:
assumes $App\ s1\ s2\ \xi \sqsubseteq t$
obtains $s1'\ s2'\ \xi'$ **where** $s1 \sqsubseteq s1'\ s2 \sqsubseteq s2'\ \xi \sqsubseteq_m \xi'\ t = App\ s1'\ s2'\ \xi'$
⟨*proof*⟩

lemma *compl-Lam-leftE*:
assumes $Lam\ x\ \zeta\ s\ \xi \sqsubseteq t$
obtains $s'\ \zeta'\ \xi'$ **where** $s \sqsubseteq s'\ \zeta \sqsubseteq_m \zeta'\ \xi \sqsubseteq_m \xi'\ t = Lam\ x\ \zeta'\ s'\ \xi'$
⟨*proof*⟩

lemma *compl-Vr-rightE*:
assumes $t \sqsubseteq Vr\ x\ \xi$
obtains ξ' **where** $\xi' \sqsubseteq_m \xi\ t = Vr\ x\ \xi'$
⟨*proof*⟩

lemma *compl-Ct-rightE*:
assumes $t \sqsubseteq Ct\ c\ \xi$
obtains ξ' **where** $\xi' \sqsubseteq_m \xi\ t = Ct\ c\ \xi'$
⟨*proof*⟩

lemma *compl-App-rightE*:
assumes $t \sqsubseteq App\ s1\ s2\ \xi$
obtains $s1'\ s2'\ \xi'$ **where** $s1' \sqsubseteq s1\ s2' \sqsubseteq s2\ \xi' \sqsubseteq_m \xi\ t = App\ s1'\ s2'\ \xi'$
⟨*proof*⟩

lemma *compl-Lam-rightE*:
assumes $t \sqsubseteq Lam\ x\ \zeta\ s\ \xi$
obtains $s'\ \zeta'\ \xi'$ **where** $s' \sqsubseteq s\ \zeta' \sqsubseteq_m \zeta\ \xi' \sqsubseteq_m \xi\ t = Lam\ x\ \zeta'\ s'\ \xi'$
⟨*proof*⟩

1.4.2 Basic properties

lemma *compl-refl*[*simp*, *intro*]: $t \sqsubseteq t$
⟨*proof*⟩

lemma *compl-poss-eq*: $t \sqsubseteq s \Longrightarrow poss\ t = poss\ s$
⟨*proof*⟩

lemma *compl-bvars-eq*: $t \sqsubseteq s \Longrightarrow bvars\ t = bvars\ s$
⟨*proof*⟩

lemma *compl-unambiguous*: $t \sqsubseteq s \Longrightarrow$ *unambiguous* $t =$ *unambiguous* s
⟨*proof*⟩

lemma *erase-compl*[*simp*, *intro*]: *erase* $t \sqsubseteq t$
⟨*proof*⟩

lemma *remove-annot-compl*:
assumes $p \in$ *poss* t
shows *remove-annot* $t\ p \sqsubseteq t$
⟨*proof*⟩

lemma *compl-trans*: $[\![s \sqsubseteq t;\ t \sqsubseteq u]\!] \Longrightarrow s \sqsubseteq u$
⟨*proof*⟩

lemma *compl-antisym*: $[\![s \sqsubseteq t;\ t \sqsubseteq s]\!] \Longrightarrow s = t$
⟨*proof*⟩

lemma *compl-mtp-of-root*: $t \sqsubseteq s \Longrightarrow$ *mtp-of* $t\ [] \sqsubseteq_m$ *mtp-of* $s\ []$
⟨*proof*⟩

1.4.3 Positions and completions

lemma *compl-mtp-of*:
assumes $t \sqsubseteq s$
shows *mtp-of* $t\ p \sqsubseteq_m$ *mtp-of* $s\ p$
⟨*proof*⟩

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

lemma *erase-compl-remove-annot*:
assumes *erase* $t \sqsubseteq s$
shows *erase* $t \sqsubseteq$ *remove-annot* $s\ p$
⟨*proof*⟩

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* $u' \bigwedge p.\ p \in$ *poss* $t \Longrightarrow$ *mtp-of* $u\ p =$ *mtp-of* $u'\ p$
shows $u = u'$
⟨*proof*⟩

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*: $\exists\,\varrho.\ u = \mathit{subst\text{-}trm}\ \varrho\ v$ **and** *fv*: *fterm v*
and *cover*: $\forall\,\alpha \in \mathit{tvars\text{-}trm}\ v.\ \exists\,p \in \mathit{poss}\ v.\ \alpha \in \mathit{tvars\text{-}mty}\ (\mathit{mtp\text{-}of}\ v\ p) \wedge \mathit{mtp\text{-}of}\ t\ p \neq \mathit{None}$
and *compl-u'*: $t \sqsubseteq u'$ **and** *fu'*: *fterm u'*
and *inst-u'*: $\exists\,\varrho'.\ u' = \mathit{subst\text{-}trm}\ \varrho'\ v$
shows $u = u'$
⟨*proof*⟩

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

lemma *compl-strict-witness*:
assumes $s' \sqsubseteq s$ $s' \neq s$
shows $\exists\,p \in \mathit{poss}\ s.\ \mathit{mtp\text{-}of}\ s'\ p = \mathit{None} \wedge \mathit{mtp\text{-}of}\ s\ p \neq \mathit{None}$
⟨*proof*⟩

Multiple completions from agreeing substitutions.

lemma *multiple-compl*:
assumes $s \sqsubseteq \mathit{subst\text{-}trm}\ \varrho\ v$
$\bigwedge\alpha\ p.\ [\![\ p \in \mathit{poss}\ v;\ \alpha \in \mathit{tvars\text{-}mty}\ (\mathit{mtp\text{-}of}\ v\ p);\ \mathit{mtp\text{-}of}\ s\ p \neq \mathit{None}\ ]\!] \Longrightarrow \varrho\ \alpha = \varrho'\ \alpha$
shows $s \sqsubseteq \mathit{subst\text{-}trm}\ \varrho'\ v$
⟨*proof*⟩

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**
$\mathit{tpOf}\ u\ p \equiv \mathit{the}\ (\mathit{mtp\text{-}of}\ u\ p)$

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

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**
$\mathit{fvars}\ (\mathit{Vr}\ x\ (\mathit{Some}\ \sigma)) = \{(x, \sigma)\}$
| $\mathit{fvars}\ (\mathit{Vr}\ x\ \mathit{None}) = \{\}$
| $\mathit{fvars}\ (\mathit{Ct}\ c\ \xi) = \{\}$
| $\mathit{fvars}\ (\mathit{App}\ t1\ t2\ \xi) = \mathit{fvars}\ t1 \cup \mathit{fvars}\ t2$
| $\mathit{fvars}\ (\mathit{Lam}\ x\ \xi\ t\ \zeta) = \{(y, \sigma) \in \mathit{fvars}\ t.\ y \neq x\}$

lemma *fvars-subst*: $\mathit{fvars}\ (\mathit{subst\text{-}trm}\ \varrho\ t) = (\lambda(x, \sigma).\ (x, \mathit{subst\text{-}ty}\ \varrho\ \sigma))\ `\ \mathit{fvars}\ t$
⟨*proof*⟩

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

inductive *wt* :: (*'tv*,*'k*,*'v*,*'c*) *trm* ⇒ *bool* (⊢ - [*50*] *50*) **where**
  wt-Vr: ⊢ *Vr x* (*Some σ*)
| *wt-Ct*: ∃ *ϱ*. *σ* = *subst-ty ϱ* (*ctpOf c*) ⟹ ⊢ *Ct c* (*Some σ*)
| *wt-App*: ⟦ ⊢ *u*; ⊢ *v*; *tpOfR u* = *Arrow* (*tpOfR v*) *σ* ⟧ ⟹ ⊢ *App u v* (*Some σ*)
| *wt-Lam*: ⟦ ⊢ *u*; ∀ *τ*. (*x*, *τ*) ∈ *fvars u* ⟶ *τ* = *σ* ⟧ ⟹ ⊢ *Lam x* (*Some σ*) *u* (*Some* (*Arrow σ* (*tpOfR u*)))

Inversion rules for *wt*.

lemma *wt-Vr-inv*: ⊢ *Vr x ξ* ⟹ ∃ *σ*. *ξ* = *Some σ*
  ⟨*proof*⟩

lemma *wt-Ct-inv*: ⊢ *Ct c* (*Some σ*) ⟹ ∃ *ϱ*. *σ* = *subst-ty ϱ* (*ctpOf c*)
  ⟨*proof*⟩

lemma *wt-App-inv*: ⊢ *App t1 t2* (*Some σ*) ⟹ ⊢ *t1* ∧ ⊢ *t2* ∧ *tpOfR t1* = *Arrow* (*tpOfR t2*) *σ*
  ⟨*proof*⟩

lemma *wt-Lam-inv*: ⊢ *Lam x* (*Some σ*) *t* (*Some τ*) ⟹ ⊢ *t* ∧ *τ* = *Arrow σ* (*tpOfR t*)
  ⟨*proof*⟩

lemma *wt-App-invE*:
  assumes ⊢ *App u1 u2 ξ*
  obtains *σ* **where** ⊢ *u1* ⊢ *u2 ξ* = *Some σ tpOfR u1* = *Arrow* (*tpOfR u2*) *σ*
  ⟨*proof*⟩

lemma *wt-Lam-invE*:
  assumes ⊢ *Lam x ξ u ζ*
  obtains *σ* **where** ⊢ *u ξ* = *Some σ ζ* = *Some* (*Arrow σ* (*tpOfR u*))
    ∀ *τ*. (*x*, *τ*) ∈ *fvars u* ⟶ *τ* = *σ*
  ⟨*proof*⟩

lemma *wt-fterm*: ⊢ *u* ⟹ *fterm u*
  ⟨*proof*⟩

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

lemma *subst-wt*:
  assumes ⊢ *v*
  shows ⊢ *subst-trm ϱ v*
  ⟨*proof*⟩

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* ⊑ *u* and ⊢ *u*).

lemma *cterm-unique-completion*:
  assumes *cterm t unambiguous t*

$t \sqsubseteq u \vdash u\ t \sqsubseteq v \vdash v$
shows $u = v$
⟨*proof*⟩

1.4.5 Well-Typed Completions

definition *is-wt-completion* :: $('tv,'k,'v,'c)\ trm \Rightarrow ('tv,'k,'v,'c)\ trm \Rightarrow bool$ **where**
is-wt-completion t u ⟷ $t \sqsubseteq u \wedge$ *fterm u* $\wedge \vdash u$

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

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

end

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

locale *signature-with-mgen* = *signature ctpOf*
for *ctpOf* :: $'c \Rightarrow ('tv,'k)\ ty$ +
assumes *mgen-exists*:
⟦ *typeable* $(t{::}('tv,'k,'v,'c)\ trm)$; *unambiguous t* ⟧ ⟹ $\exists v.$ *is-mgen t v* **and**
tyvars-inf: *infinite* $(UNIV :: 'tv\ set)$
begin

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

lemma *mgen-is-mgen*:
assumes *typeable t unambiguous t*
shows *is-mgen t* (*mgen t*)
⟨*proof*⟩

lemma *mgen-compl*:
assumes *typeable t unambiguous t*
shows $t \sqsubseteq$ *mgen t*
⟨*proof*⟩

lemma *mgen-fterm*:
assumes *typeable t unambiguous t*
shows *fterm* (*mgen t*)
⟨*proof*⟩

lemma *mgen-wt*:
assumes *typeable t unambiguous t*
shows $\vdash$ *mgen t*

⟨*proof*⟩

lemma *mgen-most-general*:
assumes *typeable t unambiguous t is-wt-completion t u*
shows ∃ ϱ. *u* = *subst-trm* ϱ (*mgen t*)
⟨*proof*⟩

1.5 Correct Printings

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

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

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

Extra tyvars yield distinct most general completions.

lemma *two-mgen*:
assumes *ua*: *unambiguous s* **and** *mg*: *is-mgen s v* **and** *extra*: *tvars-trm v* − *tvars-trm s* ≠ {}
shows ∃ *v'*. *v'* ≠ *v* ∧ *is-mgen s v'*
⟨*proof*⟩

Lift distinct completions to distinct most general completions.

lemma *lift-to-most-general*:
fixes *s* :: ‹('*tv*,'*k*,'*v*,'*c*) *trm*›
assumes *unambiguous s is-wt-completion s u is-wt-completion s u' u* ≠ *u'*
obtains *v v'* **where** ‹*v* ≠ *v'*› ‹*is-mgen s v*› ‹*is-mgen s v'*›
⟨*proof*⟩

corollary *strong-correct-printing-iff*:
assumes ‹*unambiguous s*›
shows ‹*strong-correct-printing t s* ⟷ *correct-printing t s*›
⟨*proof*⟩

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* :: $'c \Rightarrow ('tv,'k)\ ty$ +
  fixes *pickPos* :: $('tv,'k,'v,'c)\ trm \Rightarrow ('tv,'k,'v,'c)\ trm \Rightarrow nat\ list$
  assumes *compat*: $test\ v\ s\ p \Longrightarrow test\ v\ s\ (pickPos\ v\ s)$
begin

function *decrease* :: $('tv,'k,'v,'c)\ trm \Rightarrow ('tv,'k,'v,'c)\ trm \Rightarrow ('tv,'k,'v,'c)\ trm$ **where**
  $decrease\ v\ s = (if\ \exists\, p.\ test\ v\ s\ p$
    $then\ decrease\ v\ (remove\text{-}annot\ s\ (pickPos\ v\ s))$
    $else\ s)$
  ⟨*proof*⟩

termination
⟨*proof*⟩

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*⟩

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

lemma *erase-compl-decrease*:
  $erase\ t \sqsubseteq s \Longrightarrow erase\ t \sqsubseteq decrease\ v\ s$
⟨*proof*⟩

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

lemma *decrease-coverage*:
  assumes $fterm\ v\ s \sqsubseteq subst\text{-}trm\ \varrho\ v$
    $\forall\, \alpha \in tvars\text{-}trm\ v.\ \exists\, p \in poss\ v.\ \alpha \in tvars\text{-}mty\ (mtp\text{-}of\ v\ p) \wedge mtp\text{-}of\ s\ p \neq None$
  shows $\forall\, \alpha \in tvars\text{-}trm\ v.\ \exists\, p \in poss\ v.\ \alpha \in tvars\text{-}mty\ (mtp\text{-}of\ v\ p) \wedge$
    $mtp\text{-}of\ (decrease\ v\ s)\ p \neq None$
  ⟨*proof*⟩

theorem *completeness*:
  assumes *ty*: *typeable t* **and** *ua*: *unambiguous t* **and** *ct*: *cterm t*
  shows ‹*correct-printing t* (*smobla t*)›
⟨*proof*⟩

1.8 Minimality

Supporting lemmas for minimality.

After decrease, no position passes the test.

lemma *decrease-no-test*:
  $\neg$*test v* (*decrease v s*) *p*
⟨*proof*⟩

Decrease preserves unambiguity.

lemma *decrease-unambiguous*:
  unambiguous s $\Longrightarrow$ *unambiguous* (*decrease v s*)
⟨*proof*⟩

Coverage minimality.

theorem *smobla-distinct-completions*:
  assumes *ty*: *typeable t* **and** *ua*: *unambiguous t* **and** *ct*: *cterm t*
    and *ua'*: *unambiguous s'* **and** *s'-compl*: *s'* $\sqsubseteq$ *smobla t* **and** *s'-neq*: *s'* $\neq$ *smobla t*
  obtains *v v'* **where** *v* $\neq$ *v' is-mgen s' v is-mgen s' v'*
⟨*proof*⟩

corollary *minimality*:
  assumes *typeable t unambiguous t cterm t* ‹*s* $\sqsubseteq$ *smobla t*› ‹*correct-printing t s*›
  shows ‹*s* = *smobla t*›
  ⟨*proof*⟩

end

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

2 AI-Authored proof formalization of the roundtrip algorithm

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

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

2.1 Types and Type Substitutions

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

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

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

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

Type variables occurring in a type.

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

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

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

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

lemma *arrow-neq-self*: $\tau \to \tau \neq \tau$
⟨*proof*⟩

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

lemma *unique-type-match*:
  subst-ty σ_1 τ = *subst-ty* σ_2 τ ⟹ α ∈ *tvars-ty* τ ⟹ σ_1 α = σ_2 α
  ⟨*proof*⟩

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

lemma *tvars-subst-ty*: *tvars-ty* (*subst-ty* σ τ) = ⋃(*tvars-ty* ‘ σ ‘ *tvars-ty* τ)
  ⟨*proof*⟩

lemma *subst-ty-id*:
  (⋀*v*. *v* ∈ *tvars-ty* τ ⟹ σ *v* = *TVar v*) ⟹ *subst-ty* σ τ = τ
  ⟨*proof*⟩

lemma *subst-ty-agree*:
  (⋀*v*. *v* ∈ *tvars-ty* τ ⟹ σ_1 *v* = σ_2 *v*) ⟹ *subst-ty* σ_1 τ = *subst-ty* σ_2 τ
  ⟨*proof*⟩

lemma *subst-ty-compose*:
  subst-ty σ_1 (*subst-ty* σ_2 τ) = *subst-ty* (λ*v*. *subst-ty* σ_1 (σ_2 *v*)) τ
  ⟨*proof*⟩

2.2 Contexts

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

type-synonym *ctx* = *string* ⇀ *ty*

Type variables of a context.

definition *tvars-ctx* :: *ctx* ⇒ *string set* **where**
  tvars-ctx Γ = ⋃(*tvars-ty* ‘ *ran* Γ)

Applying a type substitution to a context.

definition *subst-ctx* :: (*string* ⇒ *ty*) ⇒ *ctx* ⇒ *ctx* **where**
  subst-ctx σ Γ = *map-option* (*subst-ty* σ) ∘ Γ

lemma *subst-ctx-dom* [*simp*]: *dom* (*subst-ctx* σ Γ) = *dom* Γ
  ⟨*proof*⟩

lemma *subst-ctx-app*: *x* ∈ *dom* Γ ⟹ *subst-ctx* σ Γ *x* = *Some* (*subst-ty* σ (*the* (Γ *x*)))
  ⟨*proof*⟩

lemma *subst-ctx-update*:
  subst-ctx σ (Γ(*x* ↦ τ)) = (*subst-ctx* σ Γ)(*x* ↦ *subst-ty* σ τ)
  ⟨*proof*⟩

lemma *subst-ctx-id*:
  (⋀*v*. *v* ∈ *tvars-ctx* Γ ⟹ σ *v* = *TVar v*) ⟹ *subst-ctx* σ Γ = Γ
  ⟨*proof*⟩

2.3 Terms

2.3.1 Raw Terms

Raw terms include type constraints and binder constraints as annotations.

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

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

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

| *strip* (*RConstrain t* -) = *strip t*

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

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

lemma *strip-idem* [*simp*]: *strip* (*strip t*) = *strip t*
  ⟨*proof*⟩

lemma *strip-constraint-free*: *constraint-free* (*strip t*)
  ⟨*proof*⟩

2.3.2 Annotated Terms

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

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

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

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

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

fun *erase* :: *aterm* ⇒ *raw-term* **where**
  erase (*AConst c* -) = *RConst c*
| *erase* (*AVar x* -) = *RVar x*
| *erase* (*AAbs x* - *a* -) = *RAbs x* (*erase a*)
| *erase* (*AApp* a_1 a_2 -) = *RApp* (*erase* a_1) (*erase* a_2)

lemma *erase-constraint-free*: *constraint-free* (*erase a*)
  ⟨*proof*⟩

lemma *strip-erase* [*simp*]: *strip* (*erase a*) = *erase a*
  ⟨*proof*⟩

Definition (Type Substitution on Annotated Terms).

fun *subst-aterm* :: (*string* ⇒ *ty*) ⇒ *aterm* ⇒ *aterm* **where**
  subst-aterm σ (*AConst* c τ) = *AConst* c (*subst-ty* σ τ)
| *subst-aterm* σ (*AVar* x τ) = *AVar* x (*subst-ty* σ τ)
| *subst-aterm* σ (*AAbs* x τ a τ′) = *AAbs* x (*subst-ty* σ τ) (*subst-aterm* σ a) (*subst-ty* σ τ′)
| *subst-aterm* σ (*AApp* a_1 a_2 τ) = *AApp* (*subst-aterm* σ a_1) (*subst-aterm* σ a_2) (*subst-ty* σ τ)

Key property: erasure is invariant under type substitution.

lemma *erase-subst* [*simp*]: *erase* (*subst-aterm* σ a) = *erase* a
  ⟨*proof*⟩

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

lemma *typeof-subst* [*simp*]: *typeof-aterm* (*subst-aterm* σ a) = *subst-ty* σ (*typeof-aterm* a)
  ⟨*proof*⟩

Type variables of an annotated term.

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

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

lemma *subst-aterm-agree*:
  (⋀v. v ∈ *tvars-aterm* a ⟹ σ_1 v = σ_2 v) ⟹ *subst-aterm* σ_1 a = *subst-aterm* σ_2 a
  ⟨*proof*⟩

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

lemma *subst-aterm-injective*:
  subst-aterm σ_1 a = *subst-aterm* σ_2 a ⟹ α ∈ *tvars-aterm* a ⟹ σ_1 α = σ_2 α
  ⟨*proof*⟩

lemma *subst-ty-TVar* [*simp*]: *subst-ty TVar* τ = τ
  ⟨*proof*⟩

lemma *subst-aterm-id*:
  (⋀v. v ∈ *tvars-aterm* a ⟹ σ v = *TVar* v) ⟹ *subst-aterm* σ a = a
  ⟨*proof*⟩

2.3.3 Well-Typedness

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

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

inductive *well-typed* :: (*string* ⇒ *ty*) ⇒ *ctx* ⇒ *aterm* ⇒ *bool* **where**
  wt-const: ∃ ϱ. τ = *subst-ty* ϱ (*const-type c*) ⟹
    well-typed const-type Γ (*AConst c* τ)
| *wt-var*: Γ *x* = *Some* τ ⟹
    well-typed const-type Γ (*AVar x* τ)
| *wt-abs*: *well-typed const-type* (Γ(*x* ↦ τ)) *a* ⟹
    τ′ = τ → *typeof-aterm a* ⟹
    well-typed const-type Γ (*AAbs x* τ *a* τ′)
| *wt-app*: *well-typed const-type* Γ a_1 ⟹
    well-typed const-type Γ a_2 ⟹
    typeof-aterm a_1 = *typeof-aterm* a_2 → τ ⟹
    well-typed const-type Γ (*AApp* a_1 a_2 τ)

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

lemma *subst-preserves-wt*:
  well-typed ct Γ *a* ⟹ *well-typed ct* (*subst-ctx* σ Γ) (*subst-aterm* σ *a*)
⟨*proof*⟩

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

corollary *subst-wt-identity-ctx*:
  well-typed ct Γ *a* ⟹ (⋀*v*. *v* ∈ *tvars-ctx* Γ ⟹ σ *v* = *TVar v*) ⟹
   well-typed ct Γ (*subst-aterm* σ *a*)
  ⟨*proof*⟩

2.4 Positions and Post-Order Enumeration

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

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

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

fun *enum-aterm* :: *aterm* ⇒ (*nat* × *raw-term* × *ty*) *list* **where**
  enum-aterm (*AConst c* τ) = [(*0*, *RConst c*, τ)]

| *enum-aterm* (*AVar x τ*) = [(*0*, *RVar x*, *τ*)]
| *enum-aterm* (*AAbs x τ a*$_1$ *τ′*) =
  (*let L* = *enum-aterm a*$_1$; *n* = *length L in*
   [(*0*, *RVar x*, *τ*)] @ *shift-enum 1 L* @ [(*1* + *n*, *RAbs x* (*erase a*$_1$), *τ′*)])
| *enum-aterm* (*AApp a*$_1$ *a*$_2$ *τ*) =
  (*let L*$_1$ = *enum-aterm a*$_1$; *n*$_1$ = *length L*$_1$;
       L$_2$ = *enum-aterm a*$_2$; *n*$_2$ = *length L*$_2$ *in*
   L$_1$ @ *shift-enum n*$_1$ *L*$_2$ @ [(*n*$_1$ + *n*$_2$, *RApp* (*erase a*$_1$) (*erase a*$_2$), *τ*)])

The set of positions.

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

The type at a given position in the enumeration.

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

The subterm at a given position in the enumeration.

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

2.4.1 Basic Properties of Enumeration

lemma *length-shift-enum* [*simp*]: *length* (*shift-enum k L*) = *length L*
  ⟨*proof*⟩

lemma *set-shift-enum*: *set* (*shift-enum k L*) = (λ(*p*, *s*, *τ*). (*k* + *p*, *s*, *τ*)) ‘ *set L*
  ⟨*proof*⟩

lemma *in-shift-enum-iff*:
  (*p*, *s*, *τ*) ∈ *set* (*shift-enum k L*) ⟷ (∃ *p′*. *p* = *k* + *p′* ∧ (*p′*, *s*, *τ*) ∈ *set L*)
  ⟨*proof*⟩

lemma *shift-enum-shift-enum*:
  shift-enum k$_1$ (*shift-enum k*$_2$ *L*) = *shift-enum* (*k*$_1$ + *k*$_2$) *L*
  ⟨*proof*⟩

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

fun *aterm-node-count* :: *aterm* ⇒ *nat* **where**
  aterm-node-count (*AConst* - -) = *1*
| *aterm-node-count* (*AVar* - -) = *1*
| *aterm-node-count* (*AAbs* - - *a* -) = *2* + *aterm-node-count a*
| *aterm-node-count* (*AApp a*$_1$ *a*$_2$ -) = *1* + *aterm-node-count a*$_1$ + *aterm-node-count a*$_2$

lemma *length-enum-aterm*: *length* (*enum-aterm a*) = *aterm-node-count a*

⟨*proof*⟩

Every annotated term has at least one position.

lemma *enum-aterm-nonempty*: *enum-aterm a* ≠ []
⟨*proof*⟩

lemma *aterm-node-count-pos*: *aterm-node-count a* ≥ *1*
⟨*proof*⟩

2.4.2 Substitution Distributes to Positions

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

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

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

lemma *length-map-enum-ty* [*simp*]: *length* (*map-enum-ty* σ *L*) = *length L*
⟨*proof*⟩

lemma *map-enum-ty-shift*: *map-enum-ty* σ (*shift-enum k L*) = *shift-enum k* (*map-enum-ty* σ *L*)
⟨*proof*⟩

lemma *map-enum-ty-append*: *map-enum-ty* σ (L_1 @ L_2) = *map-enum-ty* σ L_1 @ *map-enum-ty* σ L_2
⟨*proof*⟩

The main lemma: the enumeration commutes with type substitution.

lemma *enum-subst-aterm*:
enum-aterm (*subst-aterm* σ *a*) = *map-enum-ty* σ (*enum-aterm a*)
⟨*proof*⟩

corollary *type-at-pos-subst-iff*:
(*p*, *s*, τ') ∈ *set* (*enum-aterm* (*subst-aterm* σ *a*)) ⟷
(∃ τ. (*p*, *s*, τ) ∈ *set* (*enum-aterm a*) ∧ τ' = *subst-ty* σ τ)
⟨*proof*⟩

corollary *type-at-pos-subst*:
(*p*, *s*, τ) ∈ *set* (*enum-aterm a*) ⟹ (*p*, *s*, *subst-ty* σ τ) ∈ *set* (*enum-aterm* (*subst-aterm* σ *a*))
⟨*proof*⟩

The node count is invariant under type substitution.

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

aterm-node-count (*subst-aterm* σ *a*) = *aterm-node-count a*
⟨*proof*⟩

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

lemma *pos-set-subst* [*simp*]: *pos-set* (*subst-aterm* σ *a*) = *pos-set a*
⟨*proof*⟩

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

lemma *tvars-aterm-subset-enum*:
$\alpha \in$ *tvars-aterm a* $\Longrightarrow \exists\, p\ s\ \tau.\ (p,\ s,\ \tau) \in$ *set* (*enum-aterm a*) $\wedge\ \alpha \in$ *tvars-ty* τ
⟨*proof*⟩

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

lemma *enum-tvars-subset-aterm*:
$(p,\ s,\ \tau) \in$ *set* (*enum-aterm a*) $\Longrightarrow$ *tvars-ty* $\tau \subseteq$ *tvars-aterm a*
⟨*proof*⟩

2.4.3 Distinctness and Range of Position Numbers

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

lemma *map-fst-shift-enum*: *map fst* (*shift-enum k L*) = *map* ((+) *k*) (*map fst L*)
⟨*proof*⟩

lemma *map-fst-enum-aterm*: *map fst* (*enum-aterm a*) = [*0..<aterm-node-count a*]
⟨*proof*⟩

Corollary: position numbers are distinct.

lemma *distinct-enum-fst*: *distinct* (*map fst* (*enum-aterm a*))
⟨*proof*⟩

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

lemma *pos-set-range*: *pos-set a* = {*0..<aterm-node-count a*}
⟨*proof*⟩

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

lemma *enum-aterm-unique*:
assumes $(p,\ s,\ \tau) \in$ *set* (*enum-aterm a*) $(p,\ s',\ \tau') \in$ *set* (*enum-aterm a*)
shows $s = s'\ \tau = \tau'$
⟨*proof*⟩

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

2.5 The Annotation Algorithm

2.5.1 Type Inference Assumption

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

locale *annotation-problem* =
  fixes *const-type* :: *string* ⇒ *ty*
    and Γ :: *ctx*
    and *a* :: *aterm*
    and *a-star* :: *aterm*
    and *σ* :: *string* ⇒ *ty*
  assumes *a-wt*: *well-typed const-type* Γ *a*
    and *a-star-wt*: *well-typed const-type* Γ *a-star*
    and *same-erasure*: *erase a-star* = *erase a*
    and *matching*: *subst-aterm σ a-star* = *a*
    and *freshness*: ⋀α. α ∈ *tvars-ctx* Γ ⟹ *σ* α = *TVar* α
begin

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

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

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

definition *key* :: *nat* ⇒ *string set* **where**
  key p = {α. ∃ *s τ*. (*p*, *s*, *τ*) ∈ *set* (*enum-aterm a-star*) ∧ α ∈ *tvars-ty τ*} ∩ *V*

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

lemma *sigma-id-on-ctx*: α ∈ *tvars-ctx* Γ ⟹ *σ* α = *TVar* α
  ⟨*proof*⟩

Positions of *a-star* and *a* coincide.

lemma *pos-set-eq*: *pos-set a* = *pos-set a-star*
  ⟨*proof*⟩

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 α ∈ *V*

shows $\exists\, p \in \mathit{pos\text{-}set}\ \mathit{a\text{-}star}.\ \alpha \in \mathit{key}\ p$
⟨*proof*⟩

Consistency of a term a' with annotations at a set of positions P': for every position $p \in P'$, the type at position p in a' agrees with the type at position p in a (i.e., with *subst-ty* $\sigma\ \tau$ where τ is the type at position p in *a-star*). This directly follows the paper's definition: since *erase* $a' = \mathit{erase}\ \mathit{a\text{-}star}$, 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 $(p,\ s,\ \tau) \in \mathit{set}\ (\mathit{enum\text{-}aterm}\ \mathit{a\text{-}star})$.

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

lemma *consistent-withI*:
$(\bigwedge p\ s\ \tau.\ p \in P' \Longrightarrow (p,\ s,\ \tau) \in \mathit{set}\ (\mathit{enum\text{-}aterm}\ \mathit{a\text{-}star}) \Longrightarrow$
$\exists\, \tau'.\ (p,\ s,\ \tau') \in \mathit{set}\ (\mathit{enum\text{-}aterm}\ a') \wedge \tau' = \mathit{subst\text{-}ty}\ \sigma\ \tau)$
$\Longrightarrow \mathit{consistent\text{-}with}\ a'\ P'$
⟨*proof*⟩

lemma *consistent-withE*:
assumes *consistent-with* $a'\ P'\ p \in P'\ (p,\ s,\ \tau) \in \mathit{set}\ (\mathit{enum\text{-}aterm}\ \mathit{a\text{-}star})$
obtains τ' **where** $(p,\ s,\ \tau') \in \mathit{set}\ (\mathit{enum\text{-}aterm}\ a')\ \tau' = \mathit{subst\text{-}ty}\ \sigma\ \tau$
⟨*proof*⟩

lemma *consistent-with-mono*:
$\mathit{consistent\text{-}with}\ a'\ P' \Longrightarrow Q \subseteq P' \Longrightarrow \mathit{consistent\text{-}with}\ a'\ Q$
⟨*proof*⟩

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

lemma *consistent-with-substI*:
assumes $\bigwedge p\ s\ \tau.\ p \in P' \Longrightarrow (p,\ s,\ \tau) \in \mathit{set}\ (\mathit{enum\text{-}aterm}\ \mathit{a\text{-}star}) \Longrightarrow$
$\mathit{subst\text{-}ty}\ \sigma'\ \tau = \mathit{subst\text{-}ty}\ \sigma\ \tau$
shows $\mathit{consistent\text{-}with}\ (\mathit{subst\text{-}aterm}\ \sigma'\ \mathit{a\text{-}star})\ P'$
⟨*proof*⟩

lemma *consistent-with-substD*:
assumes $\mathit{consistent\text{-}with}\ (\mathit{subst\text{-}aterm}\ \sigma'\ \mathit{a\text{-}star})\ P'$
$p \in P'\ (p,\ s,\ \tau) \in \mathit{set}\ (\mathit{enum\text{-}aterm}\ \mathit{a\text{-}star})$
shows $\mathit{subst\text{-}ty}\ \sigma'\ \tau = \mathit{subst\text{-}ty}\ \sigma\ \tau$
⟨*proof*⟩

end

An annotation selection extends the annotation problem with a finite set of

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

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

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

lemma *coverage-mem*: $\alpha \in V \Longrightarrow \exists p \in P.\ \alpha \in key\ p$
  ⟨*proof*⟩

2.5.2 Annotations determine the substitution

lemma *sigma-agree-on-V*:
  assumes *agreement*: $\bigwedge p\ s\ \tau.\ p \in P \Longrightarrow (p, s, \tau) \in set\ (enum\text{-}aterm\ a\text{-}star) \Longrightarrow subst\text{-}ty\ \sigma'\ \tau = subst\text{-}ty\ \sigma\ \tau$
  and $\langle \alpha \in V \rangle$
shows $\sigma'\ \alpha = \sigma\ \alpha$
⟨*proof*⟩

The main result:

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

2.5.3 Completeness

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

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

theorem *completeness*:
  assumes *a'-wt*: *well-typed const-type* Γ a'
    and *a'-erase*: $erase\ a' = erase\ a$
    and *principality*: $\exists \sigma'.\ subst\text{-}aterm\ \sigma'\ a\text{-}star = a' \wedge (\forall \alpha \in tvars\text{-}ctx\ \Gamma.\ \sigma'\ \alpha = TVar\ \alpha)$
    and *consist*: *consistent-with* a' P
  shows $a' = a$
⟨*proof*⟩

2.5.4 Local Minimality

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

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

end

2.6 Annotation Insertion

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

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

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

The top-level annotation function.

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

The number of positions traversed equals the node count.

lemma *ins-count*: *snd* (*ins* σ *a P k*) = *aterm-node-count a*
⟨*proof*⟩

Stripping constraints from the output of ins recovers the erasure.

lemma *strip-ins*: *strip* (*fst* (*ins* σ *a P k*)) = *erase a*
⟨*proof*⟩

Corollary: annotate preserves the raw term under stripping.

corollary *strip-annotate*: *strip* (*annotate* σ *a P*) = *erase a*
⟨*proof*⟩

2.7 The Reverse-Greedy Algorithm

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

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

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

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

definition *rg-keep* :: (*string* ⇒ *nat*) ⇒ *string set* ⇒ *bool* **where**
rg-keep cnt K = (∃ $\alpha \in K$. *cnt* $\alpha \leq 1$)

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

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

Initialize the count for each variable.

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

The full algorithm.

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

2.7.1 Basic Properties of the Fold

The kept set is a subset of the candidates.

lemma *rg-fold-subset*: *fst* (*rg-fold cands cnt*) ⊆ *fst* ‘ *set cands*
⟨*proof*⟩

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

lemma *rg-fold-cnt-unchanged*:
∀(*p*, *K*) ∈ *set cands*. *α* ∉ *K* ⟹ *snd* (*rg-fold cands cnt*) *α* = *cnt α*
⟨*proof*⟩

The count never increases during the fold.

lemma *rg-fold-cnt-mono*: *snd* (*rg-fold cands cnt*) *α* ≤ *cnt α*
⟨*proof*⟩

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

lemma *rg-fold-no-kept-zero*:
assumes *no-kept*: ∀ *p K*. (*p*, *K*) ∈ *set cands* ⟶ *α* ∈ *K* ⟶ *p* ∉ *fst* (*rg-fold cands cnt*)
and *cnt-eq*: *cnt α* = *length* (*filter* (λ(-, *K*). *α* ∈ *K*) *cands*)
shows *snd* (*rg-fold cands cnt*) *α* = *0*
⟨*proof*⟩

2.7.2 Coverage

Lemma (Reverse-Greedy Preserves Full Coverage). The key invariant: the fold maintains *1* ≤ *cnt α* for every α that appears in some candidate's key. This is because: - At a keep step: count is unchanged (passes to recursive call as-is). - At a drop step: we only drop when all counts in *key p* are > *1*, so after decrementing they are ≥ *1*. For variables NOT in *key p*, the count is unchanged.

At termination, *1* ≤ *cnt α* 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 $\bigwedge\alpha$. *cnt* $\alpha \geq 1$
shows *snd* (*rg-fold cands cnt*) $\alpha \geq 1$
⟨*proof*⟩

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

lemma *rg-fold-cnt-agree*:
assumes $\forall\alpha$. $(\exists(p,\ K) \in$ *set cands*. $\alpha \in K) \longrightarrow$ *cnt1* $\alpha =$ *cnt2* α
shows *fst* (*rg-fold cands cnt1*) = *fst* (*rg-fold cands cnt2*)
and $\forall\alpha$. $(\exists(p,\ K) \in$ *set cands*. $\alpha \in K) \longrightarrow$
snd (*rg-fold cands cnt1*) α = *snd* (*rg-fold cands cnt2*) α
⟨*proof*⟩

Corollary: the fold preserves count ≥ 1 for variables in candidate keys. We prove this by using $\forall\alpha$. $(\exists(p,\ K)\in$*set ?cands*. $\alpha \in K) \longrightarrow$ *?cnt1.0* $\alpha =$ *?cnt2.0* $\alpha \Longrightarrow$ *fst* (*rg-fold ?cands ?cnt1.0*) = *fst* (*rg-fold ?cands ?cnt2.0*) $\forall\alpha$. $(\exists(p,\ K)\in$*set ?cands*. $\alpha \in K) \longrightarrow$ *?cnt1.0* $\alpha =$ *?cnt2.0* $\alpha \Longrightarrow \forall\alpha$. $(\exists(p,\ K)\in$*set ?cands*. $\alpha \in K) \longrightarrow$ *snd* (*rg-fold ?cands ?cnt1.0*) α = *snd* (*rg-fold ?cands ?cnt2.0*) α 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 $\forall\alpha$. $(\exists(p,\ K) \in$ *set cands*. $\alpha \in K) \longrightarrow$ *cnt* $\alpha \geq 1$
and $(\exists(p,\ K) \in$ *set cands*. $\alpha \in K)$
shows *snd* (*rg-fold cands cnt*) $\alpha \geq 1$
⟨*proof*⟩

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*: $\forall\alpha$. *cnt* α = *length* (*filter* (λ(-, K). $\alpha \in K$) *cands*) + *extra* α
and *mem*: $(p,\ K) \in$ *set cands*

and *kept*: $p \in$ *fst* (*rg-fold cands cnt*)
shows $\exists \alpha \in K.$ *extra* $\alpha = 0 \wedge$
$(\forall p'\ K'.\ (p', K') \in$ *set cands* $\longrightarrow p' \in$ *fst* (*rg-fold cands cnt*) $\longrightarrow p' \neq p \longrightarrow \alpha \notin K')$
⟨*proof*⟩

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

lemma *rg-fold-witness*:
assumes *distinct* (*map fst cands*)
and $\forall \alpha.$ *cnt* $\alpha =$ *length* (*filter* $(\lambda(\text{-}, K).\ \alpha \in K)$ *cands*)
and $(p, K) \in$ *set cands* **and** $p \in$ *fst* (*rg-fold cands cnt*)
shows $\exists \alpha \in K.\ \forall p' \in$ *fst* (*rg-fold cands cnt*). $p' \neq p \longrightarrow$
$(\forall K'.\ (p', K') \in$ *set cands* $\longrightarrow \alpha \notin K')$
⟨*proof*⟩

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* $(\lambda(p, s, \tau).\ (p,$ *tvars-ty* $\tau \cap V))$ (*enum-aterm a-star*)

The candidate positions have distinct first components.

lemma *candidates-distinct*: *distinct* (*map fst candidates*)
⟨*proof*⟩

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*⟩

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

lemma *rg-result-subset*: *reverse-greedy candidates* ⊆ *pos-set a-star*
⟨*proof*⟩

Key in the candidate list matches key in the locale.

lemma *candidates-key-eq*:
  assumes $(p,\ K) \in$ *set candidates*
  shows $K =$ *key* p
  ⟨*proof*⟩

Coverage property: the kept positions cover all inference variables.

lemma *rg-coverage*: $\bigcup$(*key* ‘ *reverse-greedy candidates*) = V
⟨*proof*⟩

Witness property: each kept position has a witness variable.

lemma *rg-witness*:
  assumes $p \in$ *reverse-greedy candidates*
  shows $\exists\,\alpha \in$ *key* p. $\forall\,p' \in$ *reverse-greedy candidates*. $p' \neq p \longrightarrow \alpha \notin$ *key* p'
⟨*proof*⟩

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

sublocale *annotation-selection const-type* Γ *a a-star* σ *reverse-greedy candidates*
⟨*proof*⟩

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

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

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

lemma *strip-t-out*: *strip t-out* = *erase a*
  ⟨*proof*⟩

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 $\exists\,\sigma'$. *subst-aterm* σ' *a-star* = $a' \wedge (\forall\,\alpha \in$ *tvars-ctx* Γ. $\sigma'\ \alpha =$ *TVar* $\alpha)$
    and *consistent-with* a' (*reverse-greedy candidates*)
  shows $a' = a$
  ⟨*proof*⟩

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

thm *local-minimality*

end

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

3 AI-Authored proof formalization via independence systems

3.1 Independence Systems

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

locale *indep-system* =
fixes $E :: 'a\ set$ **and** $indep :: 'a\ set \Rightarrow bool$
assumes *finite-E*: *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*) $maximal :: 'a\ set \Rightarrow bool$ **where**
$maximal\ F \longleftrightarrow indep\ F \wedge (\forall\, e \in E - F.\ \neg\ indep\ (F \cup \{e\}))$

lemma (**in** *indep-system*) *maximalI*:
assumes $indep\ F\ \bigwedge e.\ e \in E - F \Longrightarrow \neg\ indep\ (F \cup \{e\})$
shows *maximal F*
⟨*proof*⟩

lemma (**in** *indep-system*) *maximalD*:
assumes *maximal F*
shows $indep\ F\ \bigwedge e.\ e \in E - F \Longrightarrow \neg\ indep\ (F \cup \{e\})$
⟨*proof*⟩

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 $greedy\text{-}step :: ('a\ set \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
$greedy\text{-}step\ indep\ e\ F = (if\ indep\ (F \cup \{e\})\ then\ F \cup \{e\}\ else\ F)$

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

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

lemma *best-in-greedy-foldl-partial*:
assumes *n* ≤ *length es*
shows *foldl* (λ*F e*. *greedy-step indep e F*) {} (*take n es*) = *greedy-partial indep es n*
⟨*proof*⟩

lemma *best-in-greedy-eq-partial*:
best-in-greedy indep es = *greedy-partial indep es* (*length es*)
⟨*proof*⟩

3.2.1 Properties of *greedy-partial*

The partial result is monotonically increasing.

lemma *greedy-partial-mono*:
i ≤ *j* ⟹ *greedy-partial indep es i* ⊆ *greedy-partial indep es j*
⟨*proof*⟩

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

lemma *greedy-partial-skip*:
assumes *j* < *length es es* ! *j* ∉ *greedy-partial indep es* (*Suc j*)
shows ¬ *indep* (*greedy-partial indep es j* ∪ {*es* ! *j*})
⟨*proof*⟩

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

lemma *greedy-partial-subset*:
assumes *i* ≤ *length es*
shows *greedy-partial indep es i* ⊆ *set* (*take i es*)
⟨*proof*⟩

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* ≤ *length es set es* ⊆ *E*

shows *indep* (*greedy-partial indep es i*)
⟨*proof*⟩

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*⟩

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

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

- The Best-In-Greedy algorithm returns a maximal independent set.
- The annotation set system is an independence system.
- Local minimality of the kept positions.
- The witness variable corollary.

3.3 The Annotation Set System

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

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

3.3.1 Setup

locale *annotation-setup* =
fixes *S* :: *nat set*
and *V* :: *string set*
and *key* :: *nat* ⇒ *string set*
assumes *finite-S*: *finite S*
assumes *finite-V*: *finite V*

assumes *key-subset*: $\bigwedge p.\ p \in S \Longrightarrow \mathit{key}\ p \subseteq V$
assumes *S-covers-V*: $\bigcup(\mathit{key}\ `\ S) = V$
begin

Coverage

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

lemma *covers-S*: *covers S*
⟨*proof*⟩

lemma *covers-mono*:
assumes *covers Q* $Q \subseteq Q'$
shows *covers* Q'
⟨*proof*⟩

3.3.2 The Independence System

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

definition *ann-indep* :: *nat set* ⇒ *bool* **where**
$\mathit{ann\text{-}indep}\ F \longleftrightarrow F \subseteq S \wedge \mathit{covers}\ (S - F)$

3.3.3 The annotation set system is an independence system

lemma *ann-indep-empty*: *ann-indep* {}
⟨*proof*⟩

lemma *ann-indep-subset-carrier*:
$\mathit{ann\text{-}indep}\ F \Longrightarrow F \subseteq S$
⟨*proof*⟩

lemma *ann-indep-hereditary*:
assumes *ann-indep F* $F' \subseteq F$
shows *ann-indep* F'
⟨*proof*⟩

The annotation set system forms an independence system.

sublocale *indep-system S ann-indep*
⟨*proof*⟩

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 $1 < \mathit{var\text{-}count}\ F\ \alpha$ for every $\alpha \in \mathit{key}\ p$, where *var-count* α counts positions in $S - F$ covering α.

definition *var-count* :: *nat set* ⇒ *string* ⇒ *nat* **where**
var-count F α = *card* {*q* ∈ *S* − *F*. *α* ∈ *key q*}

lemma *feasibility-iff-count*:
assumes *ann-indep F p* ∈ *S* − *F*
shows *ann-indep* (*F* ∪ {*p*}) ⟷ (∀ *α* ∈ *key p*. *var-count F α* > *1*)
⟨*proof*⟩

3.5 Local Minimality

lemma *local-minimality-coverage*:
assumes *ann-indep F-star*
shows *covers* (*S* − *F-star*)
⟨*proof*⟩

lemma *local-minimality*:
assumes *greedy-maximal*: *maximal F-star*
and *p-in-P*: *p* ∈ *S* − *F-star*
shows ¬ *covers* (*S* − *F-star* − {*p*})
⟨*proof*⟩

corollary *witness-variable*:
assumes *greedy-maximal*: *maximal F-star*
and *p-in-P*: *p* ∈ *S* − *F-star*
obtains *α* **where** *α* ∈ *key p α* ∈ *V*
⋀*q*. *q* ∈ *S* − *F-star* − {*p*} ⟹ *α* ∉ *key q*
⟨*proof*⟩

3.6 Reverse Greedy Correctness

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

theorem *annotation-algorithm-correct*:
assumes *perm*: *set es* = *S* **and** *distinct*: *distinct es*
shows *covers* (*reverse-greedy′ es*)
and ⋀*p*. *p* ∈ *reverse-greedy′ es* ⟹
∃ *α* ∈ *key p*. *α* ∈ *V* ∧ (∀ *q* ∈ *reverse-greedy′ es* − {*p*}. *α* ∉ *key q*)
⟨*proof*⟩
end

context *annotation-problem*
begin

The annotation problem gives rise to an annotation set system.

lemma *finite-tvars-ty*: *finite* (*tvars-ty t*)
⟨*proof*⟩

interpretation *ann*: *annotation-setup pos-set a-star V key*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *es-distinct*: *distinct es-from-enum*
⟨*proof*⟩

lemma *reverse-greedy′-subs-pos*: ‹*ann.reverse-greedy′ es-from-enum* ⊆ *pos-set a-star*›
⟨*proof*⟩

lemma *reverse-greedy′-coverage*: ‹⋃ (*key* ‘ *ann.reverse-greedy′ es-from-enum*) = *V*›
⟨*proof*⟩

lemma *reverse-greedy′-witness*:
assumes ‹*p* ∈ *ann.reverse-greedy′ es-from-enum*›
shows ‹∃ α∈*key p*. ∀ *p′*∈*ann.reverse-greedy′ es-from-enum*. *p′* ≠ *p* ⟶ α ∉ *key p′*›
⟨*proof*⟩

interpretation *ann*: *annotation-selection*
const-type
Γ *a a-star σ* ‹*ann.reverse-greedy′ es-from-enum*›
⟨*proof*⟩

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