Principal Type Inference for the Simply Typed Lambda Calculus: A Lean Formalisation

xiaoshihou514
xiaoshihou@tutamail.com

***Abstract*—This paper presents a formalization in the Lean theorem prover of a principal-type inference algorithm for the simply typed lambda calculus. The formalization demonstrates how dependent types and inductive families in Lean can be used to encode invariants that simplify reasoning about substitution and variable occurrence.**

***Index Terms*—type inference, dependent types, simply typed lambda calculus, formalisation**

I. Introduction

The simply typed lambda calculus (STLC), introduced by Church [1] and studied algebraically by Curry [2], is the theoretical foundation of typed functional programming. A central algorithmic question is *principal type inference*: given a term t, compute a most-general typing judgement $\Gamma \vdash t : \tau$ such that every other typing judgement is an instance of it. Hindley [3] showed that principal types always exist for the STLC; Milner [4] and Damas and Milner [5] later extended the result to polymorphic λ-calculi.

Robinson's unification algorithm [6] is the computational engine underlying most type-inference procedures. Despite its importance, machine-checked proofs of its correctness remain relatively rare in the literature, partly because termination is non-obvious: the standard argument requires a well-founded measure on the number of distinct type variables together with a cardinality squeeze lemma.

This report describes a complete Lean 4 [7] formalisation of:

1) The STLC grammar (types, terms) and a type-assignment checker.
2) Principal algorithm for STLC implemented in Lean 4.
3) Termination of Robinson's unification algorithm leveraging lean's dependent type system.
4) A partial soundness result for inference.

II. Background and Definitions

A. Simply Typed Lambda Calculus

Definition 2.1.1: The set CurryType of *simple types* is generated by the grammar

$$\tau, \sigma ::= \varphi_p \mid \tau \to \sigma \tag{1}$$

where p ranges over an alphabet of *type variables* (characters) and $\to$ is right-associative function-type formation.

Definition 2.1.2: The set Term of *lambda terms* is generated by the grammar

$$t, s ::= x \mid \lambda x.t \mid t \quad s \tag{2}$$

where x ranges over a set of *term variables* (characters).

Definition 2.1.3: A *typing context* Γ is a finite list of pairs $(x, \tau) \in \mathrm{Char} \times \mathrm{CurryType}$ together with a *label* $\ell \in \mathrm{Char}$ used as a counter for allocating fresh type variables. We write $\Gamma(x)$ for the first type assigned to x in Γ, if any. The empty context is $\Gamma_\emptyset = ([], \mathrm{A})$. Context extension is written $\Gamma + (x : \tau) = ((x, \tau) :: \Gamma.\mathrm{env}, \Gamma.\mathrm{label})$.

Definition 2.1.4: The set of *free type variables* of a type τ, written $\mathrm{fv}(\tau)$, is

$$\begin{aligned} \mathrm{fv}(\varphi_p) &= \{p\} \\ \mathrm{fv}(\tau \to \sigma) &= \mathrm{fv}(\tau) \cup \mathrm{fv}(\sigma) \end{aligned} \tag{3}$$

Definition 2.1.5: Typing assignment $\Gamma \vdash t : \tau$ are defined by the three standard rules:

$$\begin{gathered} \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \\ \frac{\Gamma + (x : \tau_1) \vdash m : \tau_2}{\Gamma \vdash \lambda x.m : \tau_1 \to \tau_2} \\ \frac{\Gamma \vdash m : \tau_1 \to \tau_2 \quad \Gamma \vdash n : \tau_1}{\Gamma \vdash m \quad n : \tau_2} \end{gathered} \tag{4}$$

B. Type Substitutions

Definition 2.2.1: A *type substitution* is a function $S : \mathrm{CurryType} \to \mathrm{CurryType}$ that distributes over arrow types: $S(\tau \to \sigma) = S(\tau) \to S(\sigma)$. The *pointwise application* of S to a context Γ is $S(\Gamma) = \{(x, S(\tau)) \mid (x, \tau) \in \Gamma\}$.

Definition 2.2.2: For a type variable α and type σ, the *replacement* $\tau[\alpha := \sigma]$ substitutes σ for every free occurrence of α in τ:

$$\begin{aligned} \varphi_\alpha[\alpha := \sigma] &= \sigma \\ \varphi_\beta[\alpha := \sigma] &= \varphi_\beta \quad (\beta \neq \alpha) \\ \tau \to \tau'[\alpha := \sigma] &= \tau[\alpha := \sigma] \to \tau'[\alpha := \sigma] \end{aligned} \tag{5}$$

III. Implementation

A. The `check` Function

The *type-checker* `check` verifies whether a term follows type assignment rules. It requires the context to already contain all free variables of the term with their types. Its role in this formalisation is as the *specification* against which `pp'` is proved sound.

Definition 3.1.1:

$$\mathrm{check}(\Gamma, x) = \Gamma(x) \tag{6}$$

$$\mathrm{check}(\Gamma, \lambda x.m) =$$

$$\begin{cases} \text{none}, \text{if } \mathrm{check}(\Gamma, m) = \text{none} \\ \text{none}, \text{if } \mathrm{check}(\Gamma, m).\mathrm{isSome}, \Gamma(x) = \text{none} \\ \mathrm{some}(\tau' \to \tau), \text{if } \mathrm{check}(\Gamma, m).\mathrm{isSome}, \Gamma(x).\mathrm{isSome} \end{cases} \tag{7}$$

$$\mathrm{check}(\Gamma, m\ n) =$$

$$\begin{cases} \mathrm{some}(\tau_2), \text{if} \\ \qquad \mathrm{check}(\Gamma, m) \gg\!= \mathrm{check}(\Gamma, n) = \mathrm{some}(\tau_2) \\ \text{none}, \text{otherwise} \end{cases} \tag{8}$$

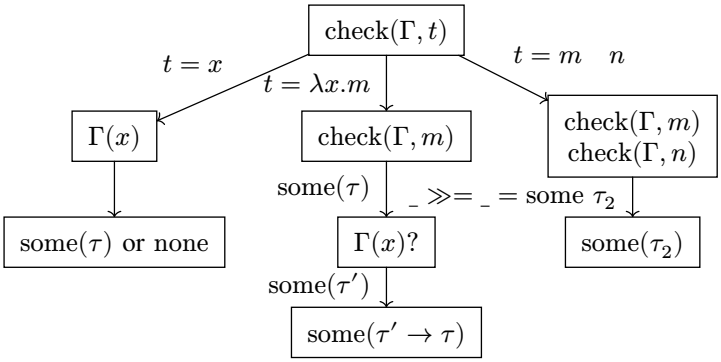

Fig. 1. Control flow of `check`.

B. `Subst`

A value $S : \mathrm{Subst}(\tau_1, \tau_2)$ packages a substitution function apply together with three propositions.

$$P_1 : \forall \tau.\ \mathrm{fv}(f(\tau)) \subseteq \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \cup \mathrm{fv}(\tau)$$

$$P_2 : \forall \tau. |\mathrm{fv}(f(\tau))| = |\mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \cup \mathrm{fv}(\tau)| \Rightarrow f = \mathrm{id} \tag{9}$$

$$P_3 : \forall a, b. f(a \to b) = f(a) \to f(b)$$

The four constructors of `Subst` and how they discharge the three invariants are summarised in Table I.

TABLE I
Subst constructors and their invariant proofs.

Constructor	P_1	P_2	P_3
`id`	Trivial	Trivial	Trivial
`replace` (α, σ)	Trivial	Lemma 4.1.6	Trivial
`replacer` (α, σ)	Lemma 4.1.3	Ex falso via Lemma 4.1.6	Trivial
`comp`(S_1, S_2)	`calc` and ind. hyp.	4.2	ind. hyp.

The `replace` and `replacer` constructors discharge P_2 by contradiction. The key auxiliary lemma (`ReplaceAddsNewVars`) shows that the cardinality equality in P_2 can *never* hold for a replacement substitution, so the implication is vacuously true. The `comp` constructor uses a cardinality squeeze: it shows that the cardinality of the image of $S_2 \circ S_1$ equals that of the combined variable set, then peels off $S_2 = \mathrm{id}$ and subsequently $S_1 = \mathrm{id}$ by two successive applications of P_2.

C. The `unify` Algorithm

Definition 3.3.1: $\mathrm{unify} : \mathrm{CurryType} \times \mathrm{CurryType} \to \mathrm{Option}(\mathrm{Subst}(\tau_1, \tau_2))$ is defined by the following case analysis. Let $\mathrm{occ}(p, \tau)$ denote the occurs-check predicate (true iff $p \in \mathrm{fv}(\tau)$).

$$\begin{aligned} &\mathrm{unify}(\varphi_p, \varphi_p) = \mathrm{some}(\mathrm{id}) \\ &\mathrm{unify}(\varphi_p, \varphi_q), p \neq q = \mathrm{some}(\mathrm{replace}(p, \varphi_q)) \\ &\mathrm{unify}(\varphi_p, \tau), \neg\, \mathrm{occ}(p, \tau) = \mathrm{some}(\mathrm{replace}(p, \tau)) \\ &\mathrm{unify}(\varphi_p, \tau), \mathrm{occ}(p, \tau) = \mathrm{none} \\ &\mathrm{unify}(\tau, \varphi_p), \neg\, \mathrm{occ}(p, \tau) = \mathrm{some}(\mathrm{replacer}(p, \tau)) \\ &\mathrm{unify}(\tau, \varphi_p), \mathrm{occ}(p, \tau) = \mathrm{none} \end{aligned} \tag{10}$$

For the arrow case, let $\mathrm{u}_1 = \mathrm{unify}(\tau, \sigma)$ and $\mathrm{u}_2 = \mathrm{unify}(S_1(\tau'), S_1(\sigma'))$:

$$\mathrm{unify}(\tau \to \tau', \sigma \to \sigma') =$$

$$\begin{cases} \text{none} \quad \text{if } \mathrm{u}_1 = \text{none} \\ \text{none} \quad \text{if } \mathrm{u}_1 = \mathrm{some}(S_1), \mathrm{u}_2 = \text{none} \\ \mathrm{some}(\mathrm{comp}(S_1, S_2)) \quad \text{if } \mathrm{u}_1 = \mathrm{some}(S_1), \mathrm{u}_2 = \mathrm{some}(S_2) \end{cases} \tag{11}$$

a) *Termination of `unify`:*
Termination of `unify` is non-trivial because the arrow case makes *two* recursive calls and the second call is on the S_1-image of the tails, not on the original tails.

Theorem 3.3.1.1 (Termination of `unify`): The function `unify` terminates under the lexicographic measure

$$\mu(\tau_1, \tau_2) = (|\mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2)|; \mathrm{sz}(\tau_1) + \mathrm{sz}(\tau_2)) \tag{12}$$

Proof: We show that each recursive call strictly decreases μ.

First call $\mathrm{unify}(\tau, \sigma)$ from the case $\mathrm{unify}(\tau \to \tau', \sigma \to \sigma')$:

$$\mathrm{fv}(\tau) \cup \mathrm{fv}(\sigma) \subseteq \mathrm{fv}(\tau \to \tau') \cup \mathrm{fv}(\sigma \to \sigma') \tag{13}$$

by definition of fv, so the cardinality does not increase. Since $\mathrm{sz}(\tau) + \mathrm{sz}(\sigma) < \mathrm{sz}(\tau \to \tau') + \mathrm{sz}(\sigma \to \sigma')$, the "pair" $(|\mathrm{fv}|, \mathrm{sz})$ decreases lexicographically (either the cardinality drops, or it stays equal while the size drops).

Second call $\mathrm{unify}(S_1(\tau'), S_1(\sigma'))$: The cardinality may or may not decrease. If it decreases strictly, we are done. Otherwise the cardinality is preserved, which

by invariant P_2 of S_1 forces $S_1 = \mathrm{id}$. When $S_1 = \mathrm{id}$, simplification gives that the size decreased. ∎

D. Context Unification `unifyCtx`

Definition 3.4.1: Let $\mathrm{dom}(\Gamma)$ denote the set of variables in the domain of Γ. Define the set of *common variable pairs*

$$C = \{(x, \Gamma_1(x), \Gamma_2(x)) \mid x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2)\} \quad (14)$$

The algorithm $\mathrm{unifyCtx}(\Gamma_1, \Gamma_2)$ computes the composed substitution

$$\mathrm{apply} := \circledcirc_{(\_,\tau_1,\tau_2)\in C}\ \mathrm{unify}(\tau_1, \tau_2) \quad (15)$$

Remark 3.4.1 (`ExSubst` and type erasure): Each $S_x = \mathrm{unify}(\tau_x, \sigma_x)$ has type $\mathrm{Subst}(\tau_x, \sigma_x)$, where the index types differ for each x. To store them in a homogeneous list, the formalisation introduces a wrapper that erased the dependent types. The function `ExSubst.cast` extracts the underlying `apply` function. This erases the type indices while retaining the distributivity property needed to prove P_3 of the composed `CtxSubst`.

E. Principal-Pair Algorithm `pp'`

Definition 3.5.1: The helper $\mathrm{pp'} : \mathrm{TypeCtx} \times \mathrm{Term} \to \mathrm{Option}(\mathrm{PrincipalPair})$ is defined by cases on the term.

Variable case. Let $(\alpha, \Gamma') = \mathrm{next}(\Gamma)$:

$$\mathrm{pp'}(\Gamma, x) = \begin{cases} \mathrm{some}(\Gamma, \tau) & \text{if } \Gamma(x) = \mathrm{some}(\tau) \\ \mathrm{some}(\Gamma' + (x : \alpha), \alpha) & \text{if } \Gamma(x) = \mathrm{none} \end{cases} \quad (16)$$

Abstraction case. Let $(\alpha, \Gamma'') = \mathrm{next}(\Gamma')$:

$$\mathrm{pp'}(\Gamma, \lambda x.m) = \begin{cases} \mathrm{none} & \text{if } \mathrm{pp'}(\Gamma, m) = \mathrm{none} \\ \mathrm{some}(\Gamma', \tau' \to \tau) & \\ \quad \text{if } \mathrm{pp'}(\Gamma, m) = \mathrm{some}(\Gamma', \tau) \text{ and } \Gamma'(x) = \mathrm{some}(\tau') & \\ \mathrm{some}(\Gamma'' + (x : \alpha), \alpha \to \tau) & \\ \quad \text{if } \mathrm{pp'}(\Gamma, m) = \mathrm{some}(\Gamma', \tau) \text{ and } \Gamma'(x) = \mathrm{none} & \end{cases} \quad (17)$$

Application case.
$(\Gamma_1, \pi_1) = \mathrm{pp'}(\Gamma, m)$,
$(\Gamma_2, \pi_2) = \mathrm{pp'}(\Gamma_1, n)$
$(\alpha, \Gamma_3) = \mathrm{next}(\Gamma_2)$,
$S_1 = \mathrm{unify}(\pi_1, \pi_2 \to \alpha)$
$S_2 = \mathrm{unifyCtx}(S_1(\Gamma_1), S_1(\Gamma_3))$, then

$$\mathrm{pp'}(\Gamma, m\ n) = \mathrm{some}(S_2(S_1(\Gamma_3)), S_2(S_1(\alpha))) \quad (18)$$

The top-level function $\mathrm{pp}(t) := \mathrm{pp'}(\Gamma_\emptyset, t)$.

IV. Structural Lemmas

This section catalogues all proven lemmas and theorems. Many are straightforward; we highlight the non-trivial ones.

A. Free Variable Lemmas

Lemma 4.1.1 (`SizeGtZ`): Every type has strictly positive size: $\forall \tau \in \mathrm{CurryType}, \mathrm{sz}(\tau) > 0$.

Lemma 4.1.2 (`VarsArrow`): $\forall \tau, \tau' \in \mathrm{CurryType}, \mathrm{fv}(\tau) \subseteq \mathrm{fv}(\tau \to \tau')$.

Lemma 4.1.3 (`ReplaceVarScope`): If $\neg\ \mathrm{occ}(\alpha, \sigma)$ then $\forall \tau, \mathrm{fv}(\tau[\alpha := \sigma]) \subseteq \{\alpha\} \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(\tau)$.

Lemma 4.1.4 (`OccursInIffInVars`): $\forall c \in \mathrm{Char}, \forall \tau \in \mathrm{CurryType}, c \in \mathrm{fv}(\tau) \Rightarrow \mathrm{occ}(c, \tau) = \mathrm{true}$.

Lemma 4.1.5 (`ReplaceElim`): If $\neg\ \mathrm{occ}(\alpha, \sigma)$ then $\forall \tau, \alpha \notin \mathrm{fv}(\tau[\alpha := \sigma])$.

Lemma 4.1.6 (`ReplaceAddsNewVars`): If $\neg\ \mathrm{occ}(\alpha, \sigma)$ then $\forall \tau$: $|\mathrm{fv}(\tau[\alpha := \sigma])| \neq |\mathrm{fv}(\varphi_\alpha) \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(\tau)|$

Proof: Suppose for contradiction that $|\mathrm{fv}(\tau[\alpha := \sigma])| = |\mathrm{fv}(\varphi_\alpha) \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(\tau)|$.

Let $L = \mathrm{fv}(\tau[\alpha := \sigma])$ and $R = \{\alpha\} \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(\tau)$.

Step 1 (strict subset). By Lemma 4.3, $L \subseteq R$. By Lemma 4.5, $\alpha \notin L$. Since $\alpha \in R$ (it belongs to $\{\alpha\} \subseteq R$), we have $L \subsetneq R$.

Step 2 (contradiction). The Mathlib lemma `Finset.eq_of_subset_of_card_le` [8] states: if $A \subseteq B$ and $|B| \leq |A|$ then $A = B$. Since $|L| = |R|$, we have $|R| \leq |L|$, so $L = R$. But $\alpha \in R$ and $\alpha \notin L$: contradiction. ∎

B. Structural Properties of `check`

Lemma 4.2.1 (`CheckInvUnderLabel`): check depends only on the `env` field of a context, not on the `label`:

$$\forall t, \Gamma.\mathrm{env} = \Gamma'.\mathrm{env} \Rightarrow \mathrm{check}(\Gamma, t) = \mathrm{check}(\Gamma', t). \quad (19)$$

Lemma 4.2.2 (`CheckInvUnderNextSubst`): Applying a substitution commutes with advancing the fresh-variable counter:

$$\mathrm{check}(\mathrm{applySubst}(f, \Gamma), m) = \mathrm{check}(\mathrm{applySubst}(f, \mathrm{next}(\Gamma).2), m) \quad (20)$$

Lemma 4.2.3 (`EnvInvUnderSubst`): applySubst preserves environment equality:

$$\Gamma.\mathrm{env} = \Gamma'.\mathrm{env} \Rightarrow (\mathrm{applySubst}(f, \Gamma)).\mathrm{env} = (\mathrm{applySubst}(f, \Gamma')).\mathrm{env}. \quad (21)$$

Lemma 4.2.4 (`LookupMap`): List lookup commutes with mapping:

$$\begin{aligned}\text{lookup}(x, \text{xys}) = \text{some}(y) \Rightarrow \\ \text{lookup}(x, \text{map}(\text{xys.map}(S)) = \text{some}(S(y)).\end{aligned} \tag{22}$$

Lemma 4.2.5 (`CheckInvUnderAdd`): If $\Gamma'(x) = \text{none}$ and $\text{ctx} = \Gamma' + (x : \alpha)$ and $m \neq \text{var } x$ and $\text{check}(\Gamma', m) = \text{some}(\pi)$, then $\text{check}(\text{ctx}, m) = \text{some}(\pi)$.

In other words, adding a binding for an unused variable to the context does not affect checking any term that does not consist solely of that variable.

Proof: By structural induction on m.

Case $m = \text{var } c$, $c \neq x$: The new context has env $(x, \alpha) :: \Gamma'.\text{env}$. Lookup proceeds: since $c \neq x$ and the `BEq` instance is `LawfulBEq`, the head is skipped, and `lookup c` falls through to $\Gamma'.\text{env}$, returning the same value.

Case $m = \lambda y.m'$: We have $\text{check}(\Gamma', m') = \text{some}(\tau)$ for some τ and $\Gamma'(y) = \text{some}(\tau')$. The induction hypothesis applies to m' if $m' \neq \text{var } x$. But if $m' = \text{var } x$ then $\text{check}(\Gamma', \text{var } x) = \Gamma'(x) = \text{none}$, contradicting the hypothesis that $\text{check}(\Gamma', m') = \text{some}(\tau)$. Thus $m' \neq \text{var } x$ and IH applies, giving $\text{check}(\text{ctx}, m') = \text{some}(\tau)$. For the lookup of y: since $x \neq y$ (if $x = y$ then $\Gamma'(x) = \text{none}$ but $\Gamma'(y) = \text{some}(\tau')$ — a contradiction since the lookup is deterministic), lookup of y in the extended context returns $\Gamma'(y) = \text{some}(\tau')$.

Case $m = m_1 \ \ m_2$: Apply IH to both m_1 and m_2 (they inherit $m \neq \text{var } x$ by a similar `check`-produces-`none`-if-only-var-is-x argument). ∎

Lemma 4.2.6 (`SubstComp`):

$$\begin{aligned}&\text{applySubst}(f, \text{applySubst}(g, \Gamma)) \\ &\quad = \text{applySubst}(f \circ g, \Gamma)\end{aligned} \tag{23}$$

Theorem 4.2.7 (`CheckSubst`): If $\text{check}(\Gamma, t) = \text{some}(\tau)$ and S distributes over arrows, then $\text{check}(\text{applySubst}(S, \Gamma), t) = \text{some}(S(\tau))$.

Proof: By structural induction on t.

Case $t = \text{var } c$: Apply Lemma 4.10 (`LookupMap`) to the hypothesis $\Gamma(c) = \tau$.

Case $t = \lambda x.m$: Let $\text{check}(\Gamma, m) = \text{some}(\tau_m)$ and $\Gamma(x) = \text{some}(\tau')$. The IH gives $\text{check}(\text{applySubst}(S, \Gamma), m) = \text{some}(S(\tau_m))$. By Lemma 4.10, $\text{applySubst}(S, \Gamma)(x) = \text{some}(S(\tau'))$. Composing: $\text{check}(...) = \text{some}(S(\tau') \to S(\tau_m)) = \text{some}(S(\tau' \to \tau_m))$ using P_3 of S.

Case $t = m \ \ n$: Let $\text{check}(\Gamma, m) = \text{some}(a \to b)$ and $\text{check}(\Gamma, n) = \text{some}(a)$ with $a == a$ (i.e. $a = a$ via `LawfulBEq`). The IH gives $\text{check}(S(\Gamma), m) = \text{some}(S(a) \to S(b))$ (using P_3 of S for the arrow) and $\text{check}(S(\Gamma), n) = \text{some}(S(a))$. Since `LawfulBEq` guarantees $S(a) == S(a)$, the branch returns $\text{some}(S(b)) = \text{some}(S(\tau))$. (The argument uses `beq_iff_eq.mp` to convert the `BEq` condition to propositional equality.) ∎

Theorem 4.2.8 (`SubstMap`): If $S_1(\tau) = S_1(a) \to S_1(b)$ then $(S_2 \circ S_1)(\tau) = (S_2 \circ S_1)(a) \to (S_2 \circ S_1)(b)$.

V. Soundness of Unification

Theorem 5.1 (`UnifySound`): If $\text{unify}(\tau_1, \tau_2) = \text{some}(S)$ then $S(\tau_1) = S(\tau_2)$.

This is the *soundness* of unification: every output of `unify` is a genuine unifier.

Proof: The proof proceeds by Lean 4's functional induction tactic `unify.induct` [7], which generates one goal per branch of `unify`'s definition.

Case $\tau_1 = \tau_2 = \varphi_p$:
$S = \text{id}$, so $S(\varphi_p) = \varphi_p = S(\varphi_p)$.

Case $\tau_1 = \varphi_p$, $\tau_2 = \varphi_q$, $p \neq q$:
$S = \text{replace}(p, \varphi_q)$. By definition of `replaceVar`, $S(\varphi_p) = \varphi_q$ and $S(\varphi_q) = \varphi_q$, so the equation holds.

Case $\tau_1 = \varphi_p$, $\tau_2 = \tau$, $\neg\ \text{occ}(p, \tau)$:
$S = \text{replace}(p, \tau)$. We need $S(\varphi_p) = S(\tau)$. $S(\varphi_p) = \tau$ by definition of `replaceVar`. For $S(\tau) = \tau$: since $\neg\ \text{occ}(p, \tau)$, the variable p does not appear in τ, so `replaceVar` acts as the identity on τ (proved by the auxiliary $\forall \sigma, \tau, \neg\ \text{occ}(p, \tau) \Rightarrow \tau[p := \sigma] = \tau$ by induction on τ). Hence both sides equal τ.

Case $\tau_1 = \varphi_p$, $\tau_2 = \tau$, $\text{occ}(p, \tau)$:
`unify` returns `none`; there is no S to consider.

Case $\tau_1 = a \to b$, $\tau_2 = c \to d$, $\text{unify}(a, c) = \text{none}$ or
$\text{unify}(S_1(b), S_1(d)) = \text{none}$: similarly `none`; no S.

Case $\tau_1 = a \to b$, $\tau_2 = c \to d$, $S = \text{comp}(S_1, S_2)$:
By induction hypothesis, $S_1(a) = S_1(c)$ and $S_2(S_1(b)) = S_2(S_1(d))$.

$$\begin{aligned}&S(a \to b) \\ &= S_2(S_1(a \to b)) \\ &= S_2(S_1(a) \to S_1(b)) \\ &= S_2(S_1(a)) \to S_2(S_1(b)) \\ &S(c \to d) \\ &= S_2(S_1(c \to d)) \\ &= S_2(S_1(c) \to S_1(d)) \\ &= S_2(S_1(c)) \to S_2(S_1(d))\end{aligned} \tag{24}$$

Since $S_1(a) = S_1(c)$ and $S_2(S_1(b)) = S_2(S_1(d))$, the two sides are equal.

Case $\tau_1 = \tau$, $\tau_2 = \varphi_p$, $\neg\,\mathrm{occ}(p, \tau)$: $S =$ replacer(p, τ), which has the same `apply` as `replace(p, τ)`. The same argument as the symmetric case above applies (using the same `replace_no_occ` auxiliary). $S(\tau) = \tau$ since $\neg$ occ(p, τ), and $S(\varphi_p) = \tau$ by definition. ■

VI. Soundness of Inference

Lemma 6.1 (`InferenceSound'`): If pp'$(\Gamma, t) = \mathrm{some}(p)$ then check$(p.\mathrm{ctx}, t) = \mathrm{some}(p.\mathrm{ty})$.

Proof: By structural induction on t.

Case $t = \mathbf{var}\ c$, $\Gamma(c) = \mathbf{some}(\tau)$: `pp'` returns (Γ, τ). `check(Γ, var c) = Γ(c) = some τ`.

Case $t = \mathbf{var}\ c$, $\Gamma(c) = \mathbf{none}$: Let $(\alpha, \Gamma') =$ next(Γ). `pp'` returns $(\Gamma' + (c : \alpha), \alpha)$. `check(Γ' + (c:α), var c) = lookup c ((c,α)::env) = some α`.

Case $t = \lambda x.m$, $\Gamma'(x) = \mathbf{some}(\tau')$: IH gives check$(\Gamma', m) = \mathrm{some}(\tau)$. `check(Γ', λx.m)` returns some$(\tau' \to \tau)$ since $\Gamma'(x) = \mathrm{some}(\tau')$.

Case $t = \lambda x.m$, $\Gamma'(x) = \mathbf{none}$: `pp'` allocates $\alpha = \mathrm{next}(\Gamma')$ and returns $(\Gamma'' + (x : \alpha), \alpha \to \tau)$. The IH gives check$(\Gamma', m) = \mathrm{some}(\tau)$. Lemma 4.11 (`CheckInvUnderAdd`) lifts this to check$(\Gamma'' + (x : \alpha), m) = \mathrm{some}(\tau)$ (since $m \neq \mathrm{var}\ x$: if $m = \mathrm{var}\ x$ then check$(\Gamma', \mathrm{var}\ x) = \Gamma'(x) = \mathrm{none}$, contradicting the IH). Lookup of x in the extended context succeeds with α.

Case $t = m\ \ n$ (application): The proof is mostly complete. It establishes the soundness of the variable and abstraction cases by applying `CheckSubst` (Theorem 4.13) and the commutativity lemmas. The final step — assembling the composed substitution in the case where check$(\mathrm{ctx}, m) = \mathrm{some}(\tau_1 \to \tau_2)$ with $\tau_1 = \tau_n$ — requires connecting UnifySound (Theorem 5.1) to the output type of `pp'`. This final connection is left as an open item (`sorry` in the current development).■

Theorem 6.2 (`InferenceSound`): If $\mathrm{pp}(t) = \mathrm{some}(p)$ then check$(p.\mathrm{ctx}, t) = \mathrm{some}(p.\mathrm{ty})$.

Proof: $\mathrm{pp}(t) = \mathrm{pp'}(\Gamma_\emptyset, t)$. Apply Lemma 6.1 directly.■

Corollary 6.2.1: If $\mathrm{pp}(t) = \mathrm{some}(p)$ then $p.\mathrm{ctx} \vdash t : p.\mathrm{ty}$ in the sense of Definition 2.5.

VII. Conclusion and Future Work

We have described an incomplete Lean 4 formalisation of principal type inference for the STLC.

The primary open item is the application case of `InferenceSound'`. The difficulty is assembling the composed substitution $S_2 \circ S_1$ from the outputs of `unify` and `unifyCtx`, and showing that the final context and type satisfy `check`. This requires a compatibility lemma relating `unifyCtx`'s action on $S_1(\Gamma_1)$ and $S_1(\Gamma_3)$ to the checking of m and n respectively.

Future directions include:

1) Complete the sorried proofs.
2) Extend the proof for more complex systems, such as Lambda calculus with names and recursion (LCNR) or Milner's ML.

VIII. AI Assistance Acknowledgement

The formalisation presented in this paper was developed with the assistance of an AI language model, specifically **DeepSeek** [9]. The AI was employed in two complementary roles:

- **As an enhanced `grind` tactic** – Many set-theoretic lemmas (e.g., subset inclusions, cardinality reasoning, and elementary facts about finite sets) were initially sketched by the AI. These proofs, while mathematically straightforward, can be tedious to write out in full detail. The AI provided high-level proof sketches that could be refined into complete Lean proofs, effectively acting as an "intelligent `grind`". This accelerated the development of lemmas such as `ReplaceVarScope`, `ReplaceElim`, and the various auxiliary facts used in the termination argument for `unify`.
- **As a proof-state summariser** – During the construction of complex inductive proofs (notably `CheckInvUnderAdd` and the application case of `InferenceSound'`), the AI was used to summarise the current hypotheses and goals. By prompting the model with the current proof context, it helped keep track of numerous side conditions and filter out the irrelevant ones.

All AI-generated proof sketches were carefully reviewed, corrected, and simplified.

References

[1] A. Church, "A Formulation of the Simple Theory of Types," *Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940.

[2] H. B. Curry, "Functionality in Combinatory Logic," *Proceedings of the National Academy of Sciences*, vol. 20, no. 11, pp. 584–590, 1934.

[3] J. R. Hindley, "The Principal Type-Scheme of an Object in Combinatory Logic," *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969.

[4] R. Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[5] L. Damas and R. Milner, "Principal Type-Schemes for Functional Programs," in *Proceedings of the 9th ACM Symposium on Principles of Programming Languages (POPL)*, 1982, pp. 207–212.

[6] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, 1965.

[7] L. de Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Proceedings of the 28th International Conference on Automated Deduction (CADE)*, 2021, pp. 625–635.

[8] The Mathlib Community, "Mathlib4: Mathematics Library for Lean 4." 2024.

[9] DeepSeek-AI, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning." [Online]. Available: https://arxiv.org/abs/2501.12948