# Principal Type Inference for the STLC: Addendum — Soundness Completion

xiaoshihou514
xiaoshihou@tutamail.com

***Abstract*—This addendum extends the Lean 4 formalisation of principal type inference for the Simply Typed Lambda Calculus described in the companion report. The primary achievement is the resolution of the application case of `InferenceSound'`, which was left as an open item in the original development. A suite of new supporting lemmas was developed, `unifyCtx` was refactored for correctness.**

***Index Terms*—type inference, Lean 4, soundness, unification**

## I. Overview

This document is an addendum to the companion report *Principal Type Inference for the Simply Typed Lambda Calculus: A Lean Formalisation*. The additions fall into three categories:

1) **Correctness fix to `unifyCtx`** (page 1): the prior implementation unified each common-variable pair independently, without threading substitutions through subsequent pairs. A new recursive helper `unifyCtx'` fixes this.
2) **New supporting lemmas and definitions** (page 1): a collection of lemmas about variable occurrence, lookup, and substitution invariants, needed to discharge the application case.
3) **Resolution of the open item in `InferenceSound'`** (page 2): the application case of the main soundness lemma is now fully proved, closing the primary open item from the companion report. `NextFresh`, previously left as a sorry'd assumption, is also now proved.

## II. Correctness Fix: Refactoring `unifyCtx`

The prior `unifyCtx` unified each common-variable pair *independently*: it mapped over the common variable list calling `unify` on each pair in isolation, then composed the resulting substitutions left-to-right. This is incorrect: the substitution from unifying the first pair should be applied to the remaining pairs before those are unified, just as Robinson's algorithm proceeds sequentially.

The new implementation introduces a recursive helper:

**Definition 2.1**:

$$\begin{aligned}&\text{unifyCtx'}([]) = \text{some}([])\\&\text{unifyCtx'}((x,a,b) :: \text{rest}) =\\&\begin{cases}\text{none},\ \text{unify}(a,b) = \text{none}\\ \text{some}([s_1] +\!+ \text{exs'}),\ \text{unify}(a,b) = \text{some}(s_1) \text{ and}\\ \qquad \text{unifyCtx'}(\text{rest.map}(s_1)) = \text{some}(\text{exs'})\end{cases}\end{aligned} \tag{1}$$

where $\text{rest.map}(s_1)$ applies $s_1$ to all remaining type pairs before recursing.

`unifyCtx` is now defined in terms of `unifyCtx'`: it collects all common variables into a list, calls `unifyCtx'`, then composes the resulting substitutions left-to-right via `foldl`.

## III. New Supporting Lemmas and Definitions

All entries in this section are new results residing in `proof.lean` unless otherwise noted.

### A. Definitions

**Definition 3.1.1** (`distributes`): $\text{distributes}(f) :=$ $\forall a, b,\quad f(a \to b) = f(a) \to f(b)$. A predicate asserting that a type function distributes over arrow types. Required by `CheckSubst` to push a substitution through function types.

**Definition 3.1.2** (`lookup_inv`): $\text{lookup\_inv}(f, \Gamma, \Gamma') :=$ $\forall x, \tau,\quad \Gamma(x) = \text{some}(\tau) \Rightarrow \Gamma'(x) = \text{some}(f(\tau))$. A predicate asserting that $f$ is a *lookup invariant* from $\Gamma$ to $\Gamma'$: if $\Gamma$ assigns $\tau$ to $x$, then $\Gamma'$ assigns $f(\tau)$ to $x$.

**Definition 3.1.3** (`check_inv`):
$\text{check\_inv}(f, \Gamma, \Gamma') := \forall t, \tau,\quad \text{check}(\Gamma, t) = \text{some}(\tau) \Rightarrow \text{check}(\Gamma', t) = \text{some}(f(\tau))$. A predicate asserting that $f$ is a *check invariant* from $\Gamma$ to $\Gamma'$.

**Definition 3.1.4** (`equiv_lookup_on`): $\text{equiv\_lookup\_on}(f, g, \Gamma) := \forall x, \tau,\quad \Gamma(x) = \text{some}(\tau) \Rightarrow g(\tau) = f(\tau)$. Two substitutions $f$ and $g$ agree on all types assigned by $\Gamma$.

**Definition**
**3.1.5** (`equiv_check_on`): equiv_check_on$(f, g, \Gamma) :=$ $\forall t, \tau,$ check$(\Gamma, t) =$ some$(\tau) \Rightarrow g(\tau) = f(\tau)$. Two substitutions $f$ and $g$ agree on all types derivable under $\Gamma$.

**Definition 3.1.6** (`occursInTerm`): The predicate occurs$(x, t)$ holds when variable $x$ appears in term $t$:

$$\begin{aligned} &\text{occurs}(x, \text{var } y) := (x = y) \\ &\text{occurs}(x, \lambda y.m) := (x = y) \vee \text{occurs}(x, m) \\ &\text{occurs}(x, m \quad n) := \text{occurs}(x, m) \vee \text{occurs}(x, n). \end{aligned} \tag{2}$$

*B. Auxiliary Lemmas*

**Theorem 3.2.1** (`LookupThenIn`): If lookup$(x, l) =$ some$(y)$ then $(x, y) \in l$.

**Theorem 3.2.2** (`FoldlInv`): If $f(a) = f(b)$, then for any list of substitutions es,

$$\begin{aligned} &(\text{es.foldl}(\lambda\ \text{acc}, \text{ex.}\ \text{ex} \circ \text{acc}, f))(a) = \\ &(\text{es.foldl}(\lambda\ \text{acc}, \text{ex.}\ \text{ex} \circ \text{acc}, f))(b). \end{aligned} \tag{3}$$

Left-folding further substitutions after $f$ preserves the equality $f(a) = f(b)$. This is a corollary of `congrArg`.

**Theorem 3.2.3** (`SubstLookup`): If $\Gamma(x) =$ some$(\tau)$ then $(\text{applySubst}(f, \Gamma))(x) = \text{some}(f(\tau))$.

**Theorem 3.2.4** (`CheckEqIfOccursInv`):
If $\forall x, \text{occurs}(x, t) \Rightarrow \Gamma_1(x) = \Gamma_2(x)$, then check$(\Gamma_1, t) =$ check$(\Gamma_2, t)$.
Type-checking of a term depends only on the types assigned to variables that *occur* in it.

**Theorem 3.2.5** (`CheckSuccessLookupSome`):
If check$(\Gamma, t) =$ some$(\tau)$, then for every variable $x$ occurring in $t$, $\exists \tau', \Gamma(x) = \text{some}(\tau')$.
Successful type-checking implies all occurring variables are bound in the context.

*C. Soundness of `unifyCtx'` and `unifyCtx`*

**Theorem 3.3.1** (`UnifyCtxSound`):
If unifyCtx'$(l) =$ some(exs) and $(x, a, b) \in l$, then letting $F =$ exs.foldl$(\lambda\ \text{acc}, \text{ex.}\ \text{ex} \circ \text{acc}, \text{id})$, we have $F(a) = F(b)$.
The composed substitution from `unifyCtx'` unifies every pair in the input list.

**Theorem 3.3.2** (`UnifyCtxMapCommon`):
If unifyCtx$(\Gamma_1, \Gamma_2) =$ some$(s)$, $\Gamma_1(x) =$ some$(\tau_1)$, and $\Gamma_2(x) =$ some$(\tau_2)$, then $s(\text{apply})(\tau_1) = s(\text{apply})(\tau_2)$.
The output substitution of `unifyCtx` equates all types assigned to common variables by the two input contexts.

*D. Invariant Lemmas for `pp'`*

**Theorem 3.4.1** (`InferenceSubsetMap`):
If pp'$(\Gamma, m) =$ some$(\langle \Gamma', \tau \rangle)$, then there exists a substitution $f$ such that $(\text{applySubst}(f, \Gamma)).\text{env} \subseteq \Gamma'.\text{env}$.
The output context contains (a substitution image of) the input context's bindings.

**Theorem 3.4.2** (`InferenceLookupInv`):
If pp'$(\Gamma, m) =$ some$(\langle \Gamma', ... \rangle)$, then there exists a substitution $f$ such that:
- lookup_inv$(f, \Gamma, \Gamma')$,
- distributes$(f)$,
- for any other $g$ with lookup_inv$(g, \Gamma, \Gamma')$, we have equiv_lookup_on$(f, g, \Gamma)$.

**Theorem 3.4.3** (`InferenceCheckInv`):
If pp'$(\Gamma, m) =$ some$(\langle \Gamma', ... \rangle)$, then there exists a substitution $f$ such that:
- check_inv$(f, \Gamma, \Gamma')$,
- distributes$(f)$,
- for any other $g$ with check_inv$(g, \Gamma, \Gamma')$ on types derivable under $\Gamma$, we have equiv_check_on$(f, g, \Gamma)$.

## IV. Resolution of open item in `InferenceSound'`

The companion report left the *application* case of the main soundness lemma as an open item (the proof contained a `sorry`). This case is now fully proved.

**Theorem 4.1** (`InferenceSound'`):
If pp'$(\Gamma, t) =$ some$(p)$ then check$(p.\text{ctx}, t) =$ some$(p.\text{ty})$.

The variable and abstraction cases were already proved in the companion report. The new work resolves the application case:
**Setting.** Suppose $t = m \quad n$. The algorithm proceeds as:

$$\begin{aligned} (\Gamma_m, \tau_1) &= \text{pp'}(\Gamma, m) \\ (\Gamma_n, \tau_2) &= \text{pp'}(\Gamma_m, n) \\ \alpha &= \text{next}(\Gamma_n).\text{fst} \\ s_1 &= \text{unify}(\tau_1, \tau_2 \to \alpha) \\ s_2 &= \text{unifyCtx}(s_1(\Gamma_m), s_1(\Gamma_n)) \\ p.\text{ctx} &= s_2(s_1(\Gamma_n)), \quad p.\text{ty} = s_2(s_1(\alpha)). \end{aligned} \tag{4}$$

The proof proceeds by cases on check$(p.\text{ctx}, m)$ and check$(p.\text{ctx}, n)$.

*Proof*:
1)

$$\begin{aligned}&\text{check pair.ctx } m\\&= \text{check (applySubst } (s_2 \circ s_1) \text{ ctx\_m) m}\end{aligned} \tag{5}$$

(via `CheckEqIfOccursInv`)

For every variable $x$ occurring in $m$, one shows $p.\text{ctx}(x) = (s_2 \circ s_1)(\Gamma_m)(x)$ using `InferenceLookupInv` (to get a canonical $g$ from $\Gamma_m$ to $\Gamma_n$) and `UnifyCtxMapCommon` (to show $s_2$ equates the $s_1$-images of corresponding types).

2)

$$\begin{aligned}&\text{check (applySubst } (s_2 \circ s_1) \text{ ctx\_m) m}\\&= \text{some } ((s_2 \circ s_1)p_1)\end{aligned} \tag{6}$$

By the induction hypothesis on $m$ together with `CheckSubst`.

3) $(s_2 \circ s_1)(\tau_1) = (s_2 \circ s_1)(\tau_2) \to (s_2 \circ s_1)(\alpha)$

From step 2, since $s_1(\tau_1) = s_1(\tau_2 \to \alpha)$ by `UnifySound`, and $s_2$ distributes over arrows.

4) $$\text{check pair.ctx (m n)} = \text{some } p.\text{ty} \tag{7}$$

By induction hypothesis on $n$ (giving check pair.ctx $n$ = some $((s_2 \circ s_1)$ p2)) and the arrow type from step 3, which ensures the argument type matches; hence the result is some $p$.ty where $p.\text{ty} = (s_2 \circ s_1)(\alpha)$.

The `if`/`else` branches where `check` returns `none` are also handled by contradiction, using the same `InferenceLookupInv` + `CheckEqIfOccursInv` machinery.

■

**Theorem 4.2** (`InferenceSound`): If $\text{pp}(t) = \text{some}(p)$ then $\text{check}(p.\text{ctx}, t) = \text{some}(p.\text{ty})$.

*Proof*: $\text{pp}(t) = \text{pp'}(\Gamma_\emptyset, t)$. Apply `InferenceSound'` directly. The theorem is now proved with no remaining `sorry`. ■

## V. NextFresh Proved

**Remark 5.1** (`NextFresh`): The assumption that `next ctx` always produces a fresh type variable is now fully proved. Formally:

$$\begin{aligned}&\text{ctx}(x) = \text{some}(\tau) \leftrightarrow\\&(\text{add}(y, \text{next}(\text{ctx}).\text{fst}, \text{next}(\text{ctx}).\text{snd}))(x) = \text{some}(\tau).\end{aligned} \tag{8}$$

The proof proceeds by unfolding `TypeCtx.lookup`, `add`, and `next` via `simp`, then closing with `grind`. The key insight is that `add` prepends a new binding for $y$, and since $x \neq y$, the lookup of $x$ is unaffected; `grind` discharges the resulting equality automatically.

## VI. Summary of Changes

TABLE I
Summary of new and modified results.

| File | Change | Status |
|---|---|---|
| `inference.lean` | `unifyCtx` refactored; `unifyCtx'` added | Complete |
| `proof.lean` | `InferenceSubsetMap` | Complete |
| `proof.lean` | `NextFresh` | Complete |
| `proof.lean` | `SubstLookup` | Complete |
| `proof.lean` | `distributes`, `lookup_inv`, `check_inv`, `equiv_lookup_on`, `equiv_check_on` | Complete |
| `proof.lean` | `InferenceLookupInv` | Complete |
| `proof.lean` | `InferenceCheckInv` | Complete |
| `proof.lean` | `occursInTerm` | Complete |
| `proof.lean` | `CheckEqIfOccursInv` | Complete |
| `proof.lean` | `CheckSuccess` `LookupSome` | Complete |
| `proof.lean` | `LookupThenIn`, `FoldlInv` | Complete |
| `proof.lean` | `UnifyCtxSound` | Complete |
| `proof.lean` | `UnifyCtxMapCommon` | Complete |
| `proof.lean` | `InferenceSound'` (application case) | ⋆ Complete |
| `proof.lean` | `InferenceSound` | ⋆ Complete |

## VII. AI Assistance Acknowledgement

The formalisation presented in this paper was developed with the assistance of an AI language model, specifically **DeepSeek** [1]. The AI was employed in two complementary roles:

1) **Initial verification of proof ideas**: Due to the originality of the proofs, I continuously propose new ideas and test their validity. AI assists me in quickly filtering out erroneous ideas and providing counterexamples, thereby saving time on manual verification.
2) **Proving intuitive results**: For certain intuitive lemmas that require detailed derivation (not directly tagged as solvable by `grind`), I use AI to generate preliminary lengthy proofs. I then refine them by removing parts that can be directly handled by automated tools like `grind`, ultimately producing rigorous proofs.

I ensure that all final submissions have been understood and reviewed by me. The use of AI is merely an auxiliary tool and does not replace my independent thinking or academic responsibility.

## References

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