Coq Proofs for Hindley–Milner Inference Soundness and Type Safety

1 Goal

This report describes two mechanized Coq developments about a small Hindley–Milner language, and states them in the usual mathematical notation so that the Coq text can be read against the rules it encodes.

1. *Inference soundness*: if the inference judgement assigns a type, the declarative judgement assigns it too. This is `HMSoundness.v`.
2. *Type safety*: a closed well-typed term is a value or can step, and stepping preserves its type. This is `HMTypeSafety.v`.

Sections 3–5 give the syntax, the rules and the proof of the first; sections 6–8 do the same for the second. Section 11 says what is not covered.

2 What this revision repaired

An earlier version of both files carried the Hindley–Milner name without the content.

Instantiation and generalization were degenerate. `HMSoundness.v` declared `inst (Forall xs T) T`, so instantiating a scheme returned its body unchanged, and `gen G T (Forall xs T)` with no side condition, so anything could be quantified. With both rules the identity, the algorithmic and the declarative relations were the same rules constructor for constructor and soundness was a restatement.

The safe language had no polymorphism. `HMTypeSafety.v` declared `ty := TInt` TArrow ty ty| — no type variables at all — with annotated lambdas and a monomorphic `let`. What was proved was the safety of the simply typed lambda calculus with `let` and addition.

Both are repaired below.

3 Inference soundness: syntax

3.1 Grammar

e	::=	$x \mid n \mid \lambda x.\, e \mid e_1\, e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$	expressions
τ	::=	$\texttt{int} \mid \alpha \mid \tau_1 \to \tau_2$	types
σ	::=	$\forall \bar{\alpha}.\, \tau$	type schemes
Γ	::=	$\cdot \mid \Gamma, x{:}\sigma$	environments

Here x and α are natural numbers, and $\bar{\alpha}$ is a list of them. A monomorphic scheme is $\forall \varepsilon.\, \tau$, written τ when no confusion arises. In Coq:

```
Inductive ty := TInt | TVar : nat -> ty | TArrow : ty -> ty -> ty.
Inductive expr := EVar : nat -> expr | EInt : nat -> expr
               | ELam : nat -> expr -> expr
               | EApp : expr -> expr -> expr
               | ELet : nat -> expr -> expr -> expr.
Inductive scheme := Sch : list nat -> ty -> scheme.
Definition env := list (nat * scheme).
```

Environments are association lists rather than functions precisely so that $\mathrm{ftv}(\Gamma)$ below is computable.

3.2 Free variables

$$\begin{array}{lcl} \mathrm{ftv}(\texttt{int}) & = & \emptyset \\ \mathrm{ftv}(\alpha) & = & \{\alpha\} \\ \mathrm{ftv}(\tau_1 \to \tau_2) & = & \mathrm{ftv}(\tau_1) \cup \mathrm{ftv}(\tau_2) \\ \mathrm{ftv}(\forall \bar{\alpha}.\, \tau) & = & \mathrm{ftv}(\tau) \setminus \bar{\alpha} \\ \mathrm{ftv}(\Gamma) & = & \bigcup_{x:\sigma \in \Gamma} \mathrm{ftv}(\sigma) \end{array}$$

In Coq these are `ftv`, `ftv_scheme`, `ftv_env`; the set difference is a `filter` over a boolean membership test `memb`, and `memb_In` relates the two.

3.3 Instantiation and generalization

A substitution s maps type variables to types and acts on types homomorphically; *s acts only on $\bar{\alpha}$* when $s(\beta) = \beta$ for every $\beta \notin \bar{\alpha}$.

$$\frac{s \text{ acts only on } \bar{\alpha}}{\forall \bar{\alpha}.\, \tau \;\preceq\; s(\tau)}\ \textsc{Inst} \qquad\qquad \frac{\bar{\alpha} \cap \mathrm{ftv}(\Gamma) = \emptyset}{\mathrm{gen}(\Gamma, \tau) \ni \forall \bar{\alpha}.\, \tau}\ \textsc{Gen}$$

Inst is the repair of the first defect: the scheme's body is really substituted into. A consequence used repeatedly is that a monomorphic scheme has exactly one instance,

$$\forall \varepsilon.\, \tau \preceq \tau' \quad \Longrightarrow \quad \tau' = \tau,$$

which is `inst_mono_inv`, proved from `only_on_nil` (a substitution acting only on ε is the identity) and `appT_id`.

Gen is the repair of the second: the Damas–Milner side condition. It is not vacuous. With $\Gamma = \{x_0 : \alpha_7\}$ the judgement $\mathrm{gen}(\Gamma, \alpha_7) \ni \forall \alpha_7.\, \alpha_7$ is *underivable*, which is `gen_rejects_env_variable`; under the old definition it was derivable.

The algorithm does not enjoy the freedom Gen allows. It commits to the maximal generalization

$$\mathrm{gen}_{\max}(\Gamma, \tau) \;=\; \forall\, (\mathrm{ftv}(\tau) \setminus \mathrm{ftv}(\Gamma)).\, \tau.$$

4 Inference soundness: rules

4.1 Declarative system

$$\frac{\Gamma(x) = \sigma \qquad \sigma \preceq \tau}{\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ \text{T-Int} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}\ \text{T-Lam}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ \text{T-App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \mathrm{gen}(\Gamma, \tau_1) \ni \sigma \qquad \Gamma, x{:}\sigma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{T-Let}$$

4.2 Algorithmic system

The rules are the same except that I-Let has no choice of scheme:

$$\frac{\Gamma \vdash_I e_1 : \tau_1 \qquad \Gamma, x{:}\mathrm{gen}_{\max}(\Gamma, \tau_1) \vdash_I e_2 : \tau_2}{\Gamma \vdash_I \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{I-Let}$$

Because T-Let carries a premise that I-Let does not, the two systems are no longer the same rules, and soundness has something to prove.

5 Inference soundness: the proof

Theorem 1 (`hm_infer_sound`)**.** *If* $\Gamma \vdash_I e : \tau$ *then* $\Gamma \vdash e : \tau$.

Shape of the proof. Induction on the derivation of $\Gamma \vdash_I e : \tau$, five cases.

I-Var and I-Int are immediate: the corresponding declarative rules have the same premises, and the instantiation premise is carried over unchanged. I-Lam and I-App follow from the induction hypotheses alone.

The one case with work is I-Let. The induction hypotheses give $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, x{:}\mathrm{gen}_{\max}(\Gamma, \tau_1) \vdash e_2 : \tau_2$. To apply T-Let one must supply its middle premise,

$$\mathrm{gen}(\Gamma, \tau_1) \ni \mathrm{gen}_{\max}(\Gamma, \tau_1),$$

that is: every variable the algorithm quantified is genuinely absent from $\mathrm{ftv}(\Gamma)$. This is the lemma `gen_max_ok`. □

Lemma 2 (`gen_max_ok`)**.** $\mathrm{gen}(\Gamma, \tau) \ni \mathrm{gen}_{\max}(\Gamma, \tau)$ *for all* Γ, τ.

Proof. Unfolding, the quantified list is $\texttt{filter}\,(\lambda a.\, \neg\,\texttt{memb}\ a\ \mathrm{ftv}(\Gamma))\ \mathrm{ftv}(\tau)$. Let a be in it. By `filter_In` the guard holds of a, so $\neg\,\texttt{memb}\ a\ \mathrm{ftv}(\Gamma) = \texttt{true}$; by `negb_true_iff`, $\texttt{memb}\ a\ \mathrm{ftv}(\Gamma) = \texttt{false}$; by `memb_false_notin`, $a \notin \mathrm{ftv}(\Gamma)$. That is exactly the side condition of Gen. □

5.1 Worked example: polymorphism is used

The scheme $\forall\alpha.\, \alpha \to \alpha$ has two instances used below, obtained from the substitutions $s_1 = [\alpha_0 := \texttt{int}]$ and $s_2 = [\alpha_0 := \texttt{int} \to \texttt{int}]$, each acting only on α_0:

$$\forall\alpha_0.\, \alpha_0 \to \alpha_0 \ \preceq\ \texttt{int} \to \texttt{int}, \qquad \forall\alpha_0.\, \alpha_0 \to \alpha_0 \ \preceq\ (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int}).$$

With these, the program

$$\texttt{let}\ id = \lambda x.\, x\ \texttt{in}\ (id\ id)\ 42$$

is derived at type `int` in the algorithmic system and transported to the declarative one by the theorem. The left occurrence of *id* is used at $(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$ and the right one at $\texttt{int} \to \texttt{int}$. The companion lemma `selfapp_needs_polymorphism` shows this is impossible with a monomorphic binding: by `inst_mono_inv` both occurrences would have to receive the same type, and those two types differ.

6 Type safety: syntax

6.1 Grammar

$$\begin{array}{lcll} t & ::= & x \mid n \mid t_1 + t_2 \mid \lambda x.\, t \mid t_1\, t_2 \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2 & \text{terms} \\ v & ::= & n \mid \lambda x.\, t & \text{values} \\ \tau & ::= & \texttt{int} \mid \alpha \mid \#i \mid \tau_1 \to \tau_2 & \text{types} \\ \sigma & ::= & \forall^n.\, \tau & \text{schemes} \end{array}$$

Lambdas carry no annotation: the language is untyped and the relation types it, as in a real HM system.

Types have *two* kinds of variable. α ranges over free type variables (`TFree`); $\#i$ is the i-th variable bound by the enclosing scheme (`TBound`), a de Bruijn index. A scheme $\forall^n.\, \tau$ binds $\#0, \ldots, \#(n-1)$ in τ.

This split is the device that keeps the development short. Instantiation replaces only $\#i$, and what it substitutes in contains only α's, so a free variable can never be captured by a quantifier. No freshness side conditions and no type substitution lemma are needed anywhere.

6.2 Instantiation

For $\bar\tau = \tau_0 \ldots \tau_{n-1}$ define

$$\begin{array}{lcl} \texttt{open}\,\bar\tau\,\texttt{int} & = & \texttt{int} \\ \texttt{open}\,\bar\tau\,\alpha & = & \alpha \\ \texttt{open}\,\bar\tau\,\#i & = & \tau_i \quad (\#i \text{ if } i \geq n) \\ \texttt{open}\,\bar\tau\,(\tau_1 \to \tau_2) & = & \texttt{open}\,\bar\tau\,\tau_1 \to \texttt{open}\,\bar\tau\,\tau_2 \end{array}$$

$$\frac{|\bar\tau| = n}{\forall^n.\, \tau \ \preceq\ \texttt{open}\,\bar\tau\,\tau}\ \textsc{Inst}$$

Since $\texttt{open}\,\varepsilon\,\tau = \tau$ unconditionally (`open_nil`), a monomorphic scheme $\forall^0.\, \tau$ again has exactly one instance, τ itself (`inst_mono`, `inst_mono_inv`).

7 Type safety: rules

7.1 Typing

$$\frac{\Gamma(x) = \sigma \qquad \sigma \preceq \tau}{\Gamma \vdash x : \tau}\ \text{S-}\textsc{Var} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ \text{S-}\textsc{Int} \qquad \frac{\Gamma \vdash t_1 : \texttt{int} \qquad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 + t_2 : \texttt{int}}\ \text{S-}\textsc{Add}$$

$$\frac{\Gamma, x{:}\forall^0.\, \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2}\ \text{S-}\textsc{Lam} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2}\ \text{S-}\textsc{App}$$

$$\frac{\begin{array}{c}\forall \bar\tau.\ |\bar\tau| = n \Rightarrow \Gamma \vdash t_1 : \texttt{open}\,\bar\tau\,\tau_0 \\ \Gamma, x{:}\forall^n.\, \tau_0 \vdash t_2 : \tau\end{array}}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \tau}\ \text{S-}\textsc{Let}$$

S-Let is generalization read semantically: rather than computing a scheme from $\mathrm{ftv}(\Gamma)$, it demands that t_1 be typeable at *every* instance of the scheme, and lets each occurrence of x in t_2 pick whichever instance it needs. The premise is a universally quantified hypothesis, which Coq accepts as a strictly positive occurrence and for which it generates the expected induction hypothesis.

7.2 Operational semantics

Call by value, left to right.

$$\frac{}{n + m \longrightarrow n{+}m}\ \text{E-Add} \qquad \frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}\ \text{E-Add1} \qquad \frac{t \longrightarrow t'}{v + t \longrightarrow v + t'}\ \text{E-Add2}$$

$$\frac{}{(\lambda x.\, t)\, v \longrightarrow t[x := v]}\ \text{E-Beta} \qquad \frac{t_1 \longrightarrow t_1'}{t_1\, t_2 \longrightarrow t_1'\, t_2}\ \text{E-App1} \qquad \frac{t \longrightarrow t'}{v\, t \longrightarrow v\, t'}\ \text{E-App2}$$

$$\frac{}{\texttt{let}\ x = v\ \texttt{in}\ t \longrightarrow t[x := v]}\ \text{E-Let} \qquad \frac{t_1 \longrightarrow t_1'}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \longrightarrow \texttt{let}\ x = t_1'\ \texttt{in}\ t_2}\ \text{E-Let1}$$

Substitution $t[x := v]$ is the usual capture-avoiding-by-shadowing definition: it stops at λx and at the body of $\texttt{let}\ x = \cdot$.

8 Type safety: the proof

8.1 Environment lemmas

Environments are functions $\mathbb{N} \to \sigma_\bot$, so the four facts needed about them are one-liners: `extend_eq`, `extend_neq`, `extend_shadow` ($\Gamma, x{:}\sigma_1, x{:}\sigma_2 = \Gamma, x{:}\sigma_2$) and `extend_permute` ($x \neq y \Rightarrow \Gamma, x{:}\sigma_x, y{:}\sigma_y = \Gamma, y{:}\sigma_y, x{:}\sigma_x$), all by case analysis on `Nat.eqb`. On top of them sit `context_invariance` (pointwise-equal environments type the same terms) and `weakening`, both by induction on the typing derivation.

8.2 Substitution

Lemma 3 (`substitution_preserves_typing`)**.** *If* $\Gamma, x{:}\sigma \vdash t : \tau$ *and* $\vdash v : \tau'$ *for* every τ' *with* $\sigma \preceq \tau'$*, then* $\Gamma \vdash t[x := v] : \tau$.

The hypothesis on v is what makes the lemma work for a polymorphic binding: v must be typeable at each instance x might be used at.

Shape of the proof. Induction on the *term* t — not on the derivation — so that each case may re-use the hypothesis on v at a different type. Six cases; the three that carry the argument are:

Variable. If $t = x$, the lookup yields σ by `extend_eq`, the typing premise gives $\sigma \preceq \tau$, and the hypothesis on v supplies $\vdash v : \tau$, which `weaken_empty` lifts to Γ. If $t = y \neq x$, `extend_neq` discards the binding and S-Var rebuilds the derivation.

Lambda. For $t = \lambda y.\, t'$ the substitution stops when $y = x$. That case is closed by `context_invariance` with `extend_shadow`: the inner binding of x hides the outer one. When $y \neq x$ the two bindings are swapped by `extend_permute` and the induction hypothesis applies.

Let. The same shadow/permute split for the body. The bound expression is handled by instantiating the universally quantified premise of S-Let at each $\bar{\tau}$ and appealing to the induction hypothesis for t_1 at that instance. □

8.3 Canonical forms

By inversion on `value` v and then on the typing derivation: a value of type `int` is an integer literal, and a value of type $\tau_1 \to \tau_2$ is a lambda. The other combinations are refuted because S-Int and S-Lam produce different type constructors.

8.4 Progress

Theorem 4 (`progress`)**.** *If* $\vdash t : \tau$ *then* t *is a value or* $t \longrightarrow t'$ *for some* t'*.*

Shape of the proof. Induction on the typing derivation, with the environment remembered as empty.

S-Var is vacuous: the empty environment yields no binding, so the premise is contradictory. S-Int and S-Lam give values directly.

S-Add: if both operands are values, canonical forms make them integer literals and E-Add fires; otherwise the first operand that steps is propagated by E-Add1 or E-Add2. S-App is the same argument, with `canonical_arrow` turning the function into a lambda so that E-Beta applies.

S-Let: the induction hypothesis for t_1 is itself universally quantified over $\bar{\tau}$, so it must be instantiated before it can be used. Any list of the right length will do, and the proof supplies `repeat int` n, whose length is n by `repeat_length`. With that, either t_1 is a value and E-Let fires, or it steps and E-Let1 propagates. □

8.5 Preservation

Theorem 5 (`preservation`)**.** *If* $\vdash t : \tau$ *and* $t \longrightarrow t'$ *then* $\vdash t' : \tau$*.*

Shape of the proof. Induction on the step derivation, inverting the typing derivation in each case. The congruence rules (E-Add1, E-Add2, E-App1, E-App2) rebuild the same rule around the induction hypothesis. E-Add is closed by S-Int. Two cases matter:

E-Beta. Inverting S-App and then S-Lam gives $\Gamma, x{:}\forall^0.\,\tau_1 \vdash t : \tau_2$ and $\vdash v : \tau_1$. The substitution lemma wants v typeable at every instance of $\forall^0.\,\tau_1$; by `inst_mono_inv` there is only τ_1, so the hypothesis at hand is enough.

E-Let. Inverting S-Let gives precisely "v is typeable at every instance of $\forall^n.\,\tau_0$" together with $\Gamma, x{:}\forall^n.\,\tau_0 \vdash t_2 : \tau$. These are the two hypotheses of the substitution lemma verbatim, and the polymorphic case closes without further work. This is the pay-off of stating S-Let semantically.

E-Let1. The bound expression steps. The new S-Let premise is obtained by instantiating the old one at each $\bar{\tau}$ and passing the result through the induction hypothesis. □

8.6 Worked example: polymorphism is used

$id = \lambda x.\, x$ is shown typeable at every instance of $\forall^1.\,\#0 \to \#0$: for any $\bar{\tau} = \tau$, `open` $[\tau]\,(\#0 \to \#0) = \tau \to \tau$, and S-Lam followed by S-Var with `inst_mono` derives $\vdash \lambda x.\, x : \tau \to \tau$ generically. That premise feeds S-Let and gives

$$\vdash \texttt{let}\ id = \lambda x.\, x\ \texttt{in}\ (id\ id)\ 42\ :\ \texttt{int},$$

with the two occurrences instantiated at [`int` $\to$ `int`] and [`int`]. The program is then stepped once by E-Let and re-typed through `preservation`.

9 Two views of generalization

The two files treat generalization from the two standard angles, and the difference is deliberate.

	`HMSoundness.v`	`HMTypeSafety.v`
form	syntactic	semantic
rule	$\bar{\alpha} \cap \mathrm{ftv}(\Gamma) = \emptyset$	typeable at every instance
binding of $\bar{\alpha}$	named	de Bruijn
environments	association lists	functions
needs $\mathrm{ftv}(\Gamma)$	yes	no

The semantic form is what lets the safety proof close without a type substitution lemma and without freshness machinery. It is a standard presentation, but it is not the syntactic Gen rule; a reader after that rule should look at `HMSoundness.v`. Relating the two — proving that the syntactic generalization satisfies the semantic condition — is the natural next step and is not done here.

10 Beyond the stated scope

Three things this report used to list as out of scope are now proved. Each is machine-checked and depends on no added axiom.

10.1 Recursion

MiniML is built on `let rec`, yet the safety proof had no recursion in it. Terms gain $\texttt{fix}\ f\ x.\, t$, a value whose own name is in scope in its body.

$$\frac{\Gamma, f{:}\forall^0.(\tau_1 \to \tau_2), x{:}\forall^0.\tau_1 \vdash t : \tau_2}{\Gamma \vdash \texttt{fix}\ f\ x.\, t : \tau_1 \to \tau_2}\ \text{S-Fix}$$

$$\frac{\mathrm{value}(v)}{(\texttt{fix}\ f\ x.\, t)\, v \longrightarrow (t[x := v])[f := \texttt{fix}\ f\ x.\, t]}\ \text{E-FixApp}$$

Every existing proof gained a case. Canonical forms at arrow type now concludes a disjunction, a function value being a lambda or a fix, and progress branches on it. The substitution lemma gains four cases rather than two, since fix has two binders and the substituted name may match either, both or neither; only in the last does the substitution enter the body. Preservation applies the substitution lemma twice for E-FixApp, peeling x to put the argument in and then f to unfold the recursion. As the environment binds x outermost, the peels come in that order and no side condition relating f and x is required.

10.2 Safety over many steps

Single-step preservation is not the statement one wants. With $\mathrm{stuck}(t) \equiv \neg\, \mathrm{value}(t) \wedge \neg\, \exists t'.\, t \longrightarrow t'$:

Theorem 6 (`soundness`)**.** *If* $\vdash t : \tau$ *and* $t \longrightarrow^* t'$ *then* $\neg\, \mathrm{stuck}(t')$.

The proof lifts preservation along the closure by induction, then applies progress at t'. Termination is deliberately not claimed: $\texttt{fix}\ f\ x.\, f\, x$ is well typed at every arrow type and steps to itself, and `loop_never_stuck` shows that running forever is not getting stuck. That distinction only becomes visible with recursion present.

10.3 The occurs check

Theorem 7 (`occurs_check_sound`)**.** *If α occurs in $\tau_1 \to \tau_2$ then no substitution s satisfies $s(\alpha) = s(\tau_1 \to \tau_2)$.*

By size. Applying any substitution to a type in which α occurs yields something at least as large as it yields on α alone, and strictly larger under an arrow, since an arrow's size exceeds the sum of its branches and every size is positive. An equation would then force $n < n$. The corollary is that $\alpha = \alpha \to \alpha$ is unsolvable — exactly the equation naive unification of $\lambda x.\, x\, x$ produces, so refusing it is what stops the algorithm from looping.

11 What is still out of scope

This is *not* a verification of the MiniML implementation. What is proved is proved about the two systems above, and the gap to the SML code in this repository is wide. The following remain future work.

- the unification procedure itself. Its occurs check is justified in Section 10, but no algorithm is defined and neither its soundness nor its completeness is proved; I-Lam still guesses τ_1 and I-App still assumes the function already has an arrow type, which is the shape unification would deliver on success;
- a principal type theorem;
- bridging the syntactic and the semantic accounts of generalization;
- lists, pairs, and pattern matching;
- correspondence with the SML implementation.

Recursion, safety over many steps and the occurs check used to be on this list. They are proved in Section 10.

11.1 A counterexample the proofs do not cover

MiniML gives the comparison operators the type $\forall\alpha.\, \alpha \times \alpha \to$ `bool`, but functional values cannot be compared at run time. The following is accepted by the type checker and then fails:

```
## let f x = x;
f : for all 'a, ('a -> 'a) = <fun>
## f = f;
Runtime error: these values cannot be compared.
```

This is a counterexample to progress for MiniML as it stands. Neither Coq file contains comparison operators, so neither rules it out; within the language each of them does describe, the theorems hold. Repairing MiniML would mean introducing equality types, as Standard ML does with `"a`, or making comparison monomorphic again.

12 Verification

The proofs were checked with the Rocq Prover 9.1.0:

```
cd coq/hm-soundness
coqc HMSoundness.v
coqc HMTypeSafety.v
```

Both compile with exit status 0 and no output beyond a deprecation warning for `From Coq`, which recent versions spell `From Stdlib`. There is no `Admitted`, no `admit` and no `Axiom` in either file; the 38 proofs are all closed with `Qed`. The main theorems were additionally audited for hidden assumptions,

```
Print Assumptions hm_infer_sound.
Print Assumptions progress.
Print Assumptions preservation.
```

and each answers `Closed under the global context`, so none of them rests on an added axiom.

The LaTeX report was built with:

```
pdflatex -interaction=nonstopmode hm_safety_report.tex
```