Hindley–Milner 型推論の健全性と型安全性

Coq による形式化と機械検証

1　目的

本稿は、小さな Hindley–Milner 言語について Coq で機械検証した二つの結果を述べる。Coq の記述と対照できるよう、文法と型付け規則を通常の数学記法で明示し、証明の進み方も具体的に記す。

1. **型推論の健全性**: 推論の判断が型を与えるなら、宣言的な判断も同じ型を与える。`HMSoundness.v` である。
2. **型安全性**: 閉じた型付き項は値であるか一歩実行でき、実行しても型が保存される。`HMTypeSafety.v` である。

第 3 節から第 5 節までが前者の文法・規則・証明、第 6 節から第 8 節までが後者である。覆っていない範囲は第 11 節に述べる。

2　今回の改訂で直したこと

以前の版は Hindley–Milner を名乗りながら、その中身を持っていなかった。

■具体化と一般化が退化していた。 `HMSoundness.v` は `inst (Forall xs T) T` と宣言していた。scheme を具体化しても本体がそのまま返る。また `gen G T (Forall xs T)` に側条件が無く、任意の変数を量化できた。両方が恒等写像なので、アルゴリズム的関係と宣言的関係は構成子ごとに同一となり、健全性の定理は言い換えに過ぎなかった。

■型安全性の対象言語に多相性が無かった。 `HMTypeSafety.v` の型は `TInt` と `TArrow` だけで、型変数が存在しなかった。ラムダは型注釈を持ち、`let` も単相であった。証明されていたのは、単純型付きラムダ計算に `let` と加算を足したものの型安全性である。

以下でどちらも修復する。

3　型推論の健全性: 文法

3.1　文法

$$
\begin{array}{lcll}
e & ::= & x \mid n \mid \lambda x.\, e \mid e_1\, e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \text{式} \\
\tau & ::= & \texttt{int} \mid \alpha \mid \tau_1 \to \tau_2 & \text{型} \\
\sigma & ::= & \forall \bar{\alpha}.\, \tau & \text{型スキーム} \\
\Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma & \text{環境}
\end{array}
$$

x と α は自然数、$\bar{\alpha}$ はその列である。単相 scheme は $\forall \varepsilon.\, \tau$ で、紛れがなければ τ と書く。Coq では次のとおり。

```
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).
```

環境を関数ではなく連想リストにしたのは、下の $\mathrm{ftv}(\Gamma)$ を計算可能にするためである。

3.2　自由型変数

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

Coq では `ftv`、`ftv_scheme`、`ftv_env` である。差集合は真偽値の所属判定 `memb` による `filter` で表し、両者を `memb_In` が結ぶ。

3.3　具体化と一般化

代入 s は型変数を型へ写し、型に準同型に作用する。$\bar{\alpha}$ に属さないすべての β について $s(\beta) = \beta$ であるとき、s は $\bar{\alpha}$ の**上でのみ作用する**という。

$$
\frac{s \text{ は } \bar{\alpha} \text{ の上でのみ作用する}}{\forall \bar{\alpha}.\, \tau \ \preceq\ s(\tau)}\ \textsc{Inst}
\qquad\qquad
\frac{\bar{\alpha} \cap \mathrm{ftv}(\Gamma) = \emptyset}{\mathrm{gen}(\Gamma, \tau) \ni \forall \bar{\alpha}.\, \tau}\ \textsc{Gen}
$$

Inst が第一の欠陥の修復である。scheme の本体が実際に置換される。ここから繰り返し使う帰結として、単相 scheme はただ一つの具体化しか持たない。

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

これが `inst_mono_inv` であり、証明は `only_on_nil`（ε の上でのみ作用する代入は恒等）と `appT_id` から出る。

Gen が第二の欠陥の修復、すなわち Damas–Milner の側条件である。空回りしていない。$\Gamma = \{x_0 : \alpha_7\}$ のとき $\mathrm{gen}(\Gamma, \alpha_7) \ni \forall \alpha_7.\, \alpha_7$ は**導出不能**であり、これが `gen_rejects_env_variable` である。以前の定義ではこれが導けた。

アルゴリズムは Gen が許す自由を持たず、最大一般化に確定する。

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

4　型推論の健全性: 規則

4.1　宣言的体系

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

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

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

4.2　アルゴリズム的体系

`let` 以外は同じ規則である。I-Let には scheme を選ぶ自由が無い。

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

T-Let は I-Let が持たない前提を一つ余分に持つ。したがって両者はもはや同じ規則ではなく、健全性には証明すべきことがある。

5　型推論の健全性: 証明の進み方

定理 1 (`hm_infer_sound`)**.** $\Gamma \vdash_I e : \tau$ ならば $\Gamma \vdash e : \tau$ である。

Proof. `intros G e T H` のあと `induction H` で $\Gamma \vdash_I e : \tau$ の導出に関する帰納法に入る。五つの場合を順に見る。

第 *1* 場合 *I-Var*。文脈に $H_1 : \Gamma(x) = \sigma$ と $H_2 : \sigma \preceq \tau$ があり、目標は $\Gamma \vdash x : \tau$。T-Var は同じ二つの前提を要求するので、`eapply TyVar; eauto` がそのまま当てて閉じる。具体化の前提はアルゴリズム側と宣言側で共通なので、書き換えは要らない。

第 *2* 場合 *I-Int*。目標 $\Gamma \vdash n : \texttt{int}$ は前提を持たない T-Int の結論そのもので、`constructor` で閉じる。

第 *3* 場合 *I-Lam*。帰納法の仮定は $IH : \Gamma, x{:}\forall\varepsilon.\tau_1 \vdash e : \tau_2$。T-Lam を `apply` すると目標がちょうど IH になる。

第 *4* 場合 *I-App*。帰納法の仮定二つを T-App の二つの前提に当てる。中間の型は `eapply` がメタ変数として置き、第一の仮定を当てた時点で確定する。

第 *5* 場合 ***I-Let*。ここが本題である。** 手元にあるのは

$$IH_1 : \Gamma \vdash e_1 : \tau_1, \qquad IH_2 : \Gamma, x{:}\mathrm{gen}_{\max}(\Gamma, \tau_1) \vdash e_2 : \tau_2$$

の二つだけで、gen についての前提は**無い**。アルゴリズム側の I-Let がそれを要求しないからである。一方 T-Let は三つの前提を持ち、その中央が

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

である。`eapply TyLet` すると $?\sigma$ はメタ変数として置かれるが、第三の前提に IH_2 を当てた時点で $?\sigma := \mathrm{gen}_{\max}(\Gamma, \tau_1)$ と確定する。したがって残る目標は

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

すなわち**アルゴリズムが計算した** *scheme* **が、宣言的体系の側条件を満たしているか**である。これは帰納法の仮定からは出ない。補題 `gen_max_ok` が要る。

証明木で書くと、この場合が何をしているかが一目で分かる。左が手元にあるアルゴリズム側の導出、右が組み立てる宣言側の導出である。二重線は帰納法の仮定で移した部分、太字が新たに証明する部分を表す。

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

$$\frac{\dfrac{}{\Gamma \vdash e_1 : \tau_1}\ IH_1 \qquad \dfrac{}{\mathrm{gen}(\Gamma, \tau_1) \ni \mathrm{gen}_{\max}(\Gamma, \tau_1)}\ \mathbf{gen\_max\_ok} \qquad \dfrac{}{\Gamma, x{:}\mathrm{gen}_{\max}(\Gamma, \tau_1) \vdash e_2 : \tau_2}\ IH_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{T-Let}$$

左の導出には前提が二つしかないのに、右は三つ要る。その差が `gen_max_ok` である。

以前の版では I-Let も T-Let と同じ gen の前提を持っており、両者の木は形まで同一であった。だから右の木は左の木のラベルを付け替えるだけで作れ、証明に義務が生じなかった。アルゴリズム側から選択の自由を取り上げたことで、初めてここに埋めるべき穴が現れる。 □

補題 2 (`gen_max_ok`)**.** すべての Γ, τ について $\mathrm{gen}(\Gamma, \tau) \ni \mathrm{gen}_{\max}(\Gamma, \tau)$。

Proof. `unfold gen_max` で量化される列が `filter` $(\lambda a. \neg\, \texttt{memb}\ a\ \mathrm{ftv}(\Gamma))\ \mathrm{ftv}(\tau)$ であることを出し、Gen を `apply` する。残る目標は、その列の任意の要素 a について $a \notin \mathrm{ftv}(\Gamma)$ を示すことである。

a が列に属するという仮定に `filter_In` を適用すると、二つの情報に分かれる。$a \in \mathrm{ftv}(\tau)$ であること（使わない）と、filter の述語が a について `true` を返すこと、すなわち

$$\neg\,(\mathtt{memb}\ a\ \mathrm{ftv}(\Gamma)) = \mathtt{true}$$

である。これに `negb_true_iff` を適用して `memb` a $\mathrm{ftv}(\Gamma)$ $=$ `false` を得る。最後に `memb_false_notin`（真偽値の所属判定が `false` なら命題としての所属も成り立たない、という補題で、`memb_In` から出る）を適用すれば $a \notin \mathrm{ftv}(\Gamma)$ が得られる。これがちょうど Gen の側条件である。

真偽値としての判定と命題としての所属を行き来する二段(`negb_true_iff` と `memb_false_notin`)が必要になるのは、ftv を計算可能にするために環境を連想リストにし、所属を真偽値で書いたことの代償である。 □

5.1 実例: 多相性が実際に使われる

scheme $\forall\alpha.\,\alpha \to \alpha$ は、α_0 の上でのみ作用する代入 $s_1 = [\alpha_0 := \mathtt{int}]$ と $s_2 = [\alpha_0 := \mathtt{int} \to \mathtt{int}]$ から、二つの具体化を得る。

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

これを用いて、プログラム

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

をアルゴリズム的体系で型 `int` に導き、定理で宣言的体系へ移す。左の id は $(\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})$ で、右の id は $\mathtt{int} \to \mathtt{int}$ で使われる。補題 `selfapp_needs_polymorphism` は、単相束縛ではこれが不可能であることを示す。`inst_mono_inv` により二つの出現は同じ型を受け取らねばならず、その二つの型は異なるからである。

6 型安全性: 文法

6.1 文法

t	::=	$x \mid n \mid t_1 + t_2 \mid \lambda x.\,t \mid t_1\,t_2 \mid \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2$	項
v	::=	$n \mid \lambda x.\,t$	値
τ	::=	$\mathtt{int} \mid \alpha \mid \#i \mid \tau_1 \to \tau_2$	型
σ	::=	$\forall^n.\,\tau$	スキーム

ラムダは型注釈を持たない。言語は型無しで、関係がそれに型を付ける。実際の HM 体系と同じ形である。

型変数は**二種類**ある。α は自由型変数（`TFree`)、$\#i$ は囲む scheme が束縛する i 番目の変数(`TBound`)、すなわち de Bruijn 指標である。scheme $\forall^n.\,\tau$ は τ の中の $\#0, \ldots, \#(n-1)$ を束縛する。

この二分割が、開発を短く保っている仕掛けである。具体化は $\#i$ しか置き換えず、置き換えて入る型には α しか現れないので、自由変数が量化子に捕獲されることが原理的に起こらない。新鮮性の側条件も型代入補題も、どこにも必要ない。

6.2 具体化

$\bar{\tau} = \tau_0 \ldots \tau_{n-1}$ に対して次を定める。

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

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

$\mathtt{open}\,\varepsilon\,\tau = \tau$ が無条件に成り立つ（`open_nil`）ので、単相 scheme $\forall^0.\,\tau$ はやはり τ ただ一つの具体化を持つ（`inst_mono`、`inst_mono_inv`）。

7 型安全性: 規則

7.1 型付け

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

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

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

S-Let は一般化を意味論的に読んだものである。$\mathrm{ftv}(\Gamma)$ から scheme を計算するのではなく、t_1 が scheme の**全ての**具体化で型付くことを要求し、t_2 の中の x の各出現が必要な具体化を選べるようにする。この前提は全称量化された仮定だが、Coq はこれを強正値の出現として受理し、期待どおりの帰納法の仮定を生成する。

7.2 操作意味論

値呼び、左から右。

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

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

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

代入 $t[x := v]$ は、λx と `let` $x = \cdot$ の本体で止まる、隠蔽による捕獲回避の通常の定義である。

8 型安全性: 証明の進み方

8.1 環境についての補題

環境は $\mathbb{N} \to \sigma_\perp$ の関数なので、必要な四つの事実は一行ずつで済む。`extend_eq`、`extend_neq`、`extend_shadow` $(\Gamma, x{:}\sigma_1, x{:}\sigma_2 = \Gamma, x{:}\sigma_2)$、`extend_permute` $(x \neq y \Rightarrow \Gamma, x{:}\sigma_x, y{:}\sigma_y = \Gamma, y{:}\sigma_y, x{:}\sigma_x)$ で、いずれも `Nat.eqb` の場合分けで終わる。その上に `context_invariance`（各点で等しい環境は同じ項に同じ型を与える）と `weakening` が乗り、どちらも型付け導出に関する帰納法である。

8.2 代入補題

補題 3 (`substitution_preserves_typing`)**.** $\Gamma, x{:}\sigma \vdash t : \tau$ であり、かつ $\sigma \preceq \tau'$ なるすべての τ' について $\vdash v : \tau'$ であるなら、$\Gamma \vdash t[x := v] : \tau$ である。

v への仮定が、多相束縛でこの補題を働かせている。x が使われうる各具体化で v が型付いていなければならない。

Proof. τ と Γ を `generalize dependent` してから、導出ではなく**項** t に関する帰納法に入る。項に関する帰納法を取るのは、各場合が v への仮定を**別の型で**使い直せるようにするためである。導出に関する帰納法だと、v の型が一つに固定されてしまい多相の場合が通らない。各場合ではまず `simpl` で `subst` を一段展開し、`inversion Ht` で型付け導出を分解する。

第 *1* 場合 $t = y$**（変数）。** `inversion` から $H_l : (\Gamma, x{:}\sigma)(y) = \texttt{Some}\ \sigma_0$ と $H_i : \sigma_0 \preceq \tau$ が出る。`Nat.eqb` $x\ y$ で場合分けする。

`true` の枝では $x = y$ なので、H_l に `extend_eq` を書き換えると `Some` $\sigma =$ `Some` σ_0 となり、`inversion` で $\sigma_0 = \sigma$。すると H_i は $\sigma \preceq \tau$ になる。v への仮定は「σ の**全ての**具体化 τ' について $\vdash v : \tau'$」であったから、H_i を渡して $\vdash v : \tau$ を得る。あとは `weaken_empty` で Γ へ持ち上げる。仮定が「全ての具体化について」の形であることが、ここで効いている。

`false` の枝では $x \neq y$。`extend_neq` で束縛を捨て、S-Var を H_l と H_i で組み直す。

第 *2* 場合 $t = n$**。** S-Int が前提なしに与える。

第 *3* 場合 $t = t_1 + t_2$**。** S-Add を組み直し、両辺に帰納法の仮定を v への仮定とともに当てる。

第 4 場合 $t = \lambda y.\, t'$。`inversion` から $H_b : (\Gamma, x{:}\sigma), y{:}\forall^0.\tau_1 \vdash t' : \tau_2$。`Nat.eqb` x y で場合分けする。

$y = x$ のとき、`subst` の定義により代入はラムダの内側へ入らない。目標は $\Gamma \vdash \lambda x.\, t' : \tau_1 \to \tau_2$ で、S-Lam の後に残るのは $\Gamma, x{:}\forall^0.\tau_1 \vdash t' : \tau_2$。手元の H_b は $(\Gamma, x{:}\sigma), x{:}\forall^0.\tau_1$ の下だが、`extend_shadow` が

$$(\Gamma, x{:}\sigma), x{:}\forall^0.\tau_1 \;=\; \Gamma, x{:}\forall^0.\tau_1$$

を各点で与えるので、`context_invariance` で環境を移せる。

$y \neq x$ のとき、代入は内側へ入る。帰納法の仮定は環境が $(\Gamma, y{:}\forall^0.\tau_1), x{:}\sigma$ の形であることを求めるが、H_b は順序が逆である。`extend_permute`（副条件 $x \neq y$）で入れ替えてから仮定を当てる。二つの枝を証明木で並べると次のようになる。

$$\dfrac{\dfrac{(\Gamma, x{:}\sigma), x{:}\forall^0.\tau_1 \vdash t' : \tau_2}{\Gamma, x{:}\forall^0.\tau_1 \vdash t' : \tau_2}\ \textsc{context\_invariance} + \textsc{extend\_shadow}}{\Gamma \vdash \lambda x.\, t' : \tau_1 \to \tau_2}\ \text{S-Lam} \qquad \dfrac{\dfrac{(\Gamma, x{:}\sigma), y{:}\forall^0.\tau_1 \vdash t' : \tau_2}{(\Gamma, y{:}\forall^0.\tau_1), x{:}\sigma \vdash t' : \tau_2}\ \textsc{context\_invariance} + \textsc{extend\_permute} \quad IH}{\Gamma, y{:}\forall^0.\tau_1 \vdash t'[x := v] : \tau_2}}{\Gamma \vdash \lambda y.\, t'[x := v] : \tau_1 \to \tau_2}\ \text{S-Lam}$$

左が $y = x$（代入が止まる）、右が $y \neq x$（代入が入る）である。どちらも S-Lam の下で環境を一段付け替えており、違いは付け替えに使う補題が `extend_shadow` か `extend_permute` か、そして帰納法の仮定を挟むかどうかだけである。

第 5 場合 $t = t_1\, t_2$。S-App を `eapply` して両辺に帰納法の仮定を当てる。

第 6 場合 $t = \texttt{let}\ y = t_1\ \texttt{in}\ t_2$。S-Let を `eapply` すると二つの目標に分かれる。

第一の目標は「代入後の t_1 が scheme の**全ての**具体化で型付く」ことである。$\bar{\tau}$ を導入し、S-Let の全称量化された前提をその $\bar{\tau}$ で具体化して $(\Gamma, x{:}\sigma) \vdash t_1 : \texttt{open}\,\bar{\tau}\,\tau_0$ を取り出し、t_1 の帰納法の仮定に渡す。前提が全称量化されているので、$\bar{\tau}$ ごとに一度ずつ帰納法の仮定を使い直すことになる。

第二の目標は本体で、第 4 場合と同じ二分岐である。$y = x$ なら `extend_shadow`、$y \neq x$ なら `extend_permute` を `context_invariance` と組み合わせる。

全体を通して働いているのは、**束縛が名前を隠すか順序を入れ替えられるか**という二つの事実と、v **への仮定が全ての具体化について述べられている**ことである。後者が多相 `let` を通している。 □

8.3 標準形の補題

`value` v と型付け導出の両方を inversion して得る。`int` 型の値は整数リテラル、$\tau_1 \to \tau_2$ 型の値はラムダである。他の組み合わせは、S-Int と S-Lam が異なる型構成子を作ることから反駁される。

8.4 Progress

定理 4 (`progress`)**.** $\vdash t : \tau$ ならば、t は値であるか、ある t' へ $t \longrightarrow t'$ と進む。

Proof. `remember empty as G` で環境を $\emptyset$ に固定してから型付け導出に関する帰納法に入る。この `remember` が要るのは、S-Lam と S-Let の部分導出が拡張された環境の下にあり、そのままでは帰納法の仮定を空環境について使えないからである。六つの場合を順に見る。

S-Var。前提は $\emptyset(x) = \texttt{Some}\ \sigma$ だが、$\emptyset$ は定義上すべての x に `None` を返す。構成子が違うので `discriminate` でこの場合ごと消える。閉じた項に自由変数は無い、という事実がここに現れる。

S-Int、*S-Lam*。それぞれ V-Int、V-Lam により値。左の選択肢を取る。

S-Add。右の選択肢を主張し、t_1 の帰納法の仮定で場合分けする。t_1 が進むなら E-Add1。t_1 が値なら t_2 の仮定で分け、t_2 が進むなら E-Add2（t_1 が値であることがその前提として要る）。両方値なら `canonical_int` を二度使って $t_1 = n$、$t_2 = m$ を取り出し、E-Add が発火する。証人は $\texttt{TmInt}\,(n + m)$。

S-App。同じ運びで、両方値のときに `canonical_arrow` を t_1 に適用して $t_1 = \lambda x.\, t$ を取り出す。これで E-Beta が使え、証人は $t[x := t_2]$。ここが標準形の補題を要する箇所で、「型が付いているのに関数の位置に関数でないものが来る」という詰まり方が起こらないことを保証している。

S-Let。**ここだけ手当てが要る。**t_1 についての帰納法の仮定は、S-Let の前提が全称量化されているために、それ自身

$$\forall \bar{\tau}.\ |\bar{\tau}| = n \Rightarrow \big(t_1 \text{ は値であるか一歩進む}\big)$$

という形をしている。使う前に $\bar{\tau}$ を一つ与えねばならない。長さが n でさえあれば結論は $\bar{\tau}$ に依らないので、どれでもよい。証明は `repeat int` n を渡し、その長さが n であることを `repeat_length` で示す。これで仮定が使える形になり、t_1 が値なら E-Let、進むなら E-Let1 が働く。

どの場合でも「値である」か「一歩進む証人を作る」かのいずれかが得られた。 □

8.5 Preservation

定理 5 (`preservation`)**.** $\vdash t : \tau$ かつ $t \longrightarrow t'$ ならば $\vdash t' : \tau$ である。

Proof. τ を `generalize dependent` してから step の導出に関する帰納法に入り、各場合でまず型付け導出を `inversion` する。step の構成子は八つ。

E-Add。`inversion` から $\tau = \texttt{int}$。目標 $\vdash n+m : \texttt{int}$ は S-Int が与える。

E-Add1、*E-Add2*。S-Add を組み直し、進む側に帰納法の仮定を当て、他方はそのまま持ち越す。

E-Beta。S-App の `inversion` で $\vdash \lambda x.\, t : \tau_1 \to \tau$ と $\vdash v : \tau_1$ を得、続いて前者を S-Lam で `inversion` して $\emptyset, x{:}\forall^0.\tau_1 \vdash t : \tau$ を取り出す。代入補題は「v が $\forall^0.\tau_1$ の**全ての**具体化で型付く」ことを求めるが、`inst_mono_inv` により具体化は τ_1 ただ一つなので、手元の $\vdash v : \tau_1$ で足りる。ラムダ束縛が単相であることが、ここで全称量化された仮定を一つの型へ潰している。

$$\dfrac{\dfrac{\dfrac{\vdash (\lambda x.\, t)\, v : \tau}{\vdash \lambda x.\, t : \tau_1 \to \tau}\ \text{S-App の inversion} \qquad \vdash v : \tau_1}{\emptyset, x{:}\forall^0.\tau_1 \vdash t : \tau}\ \text{S-Lam の inversion} \qquad \forall \tau'.\ \forall^0.\tau_1 \preceq \tau' \Rightarrow \vdash v : \tau'}{\vdash t[x := v] : \tau}\ \text{代入補題}\ \text{inst\_mono\_inv}$$

E-App1、*E-App2*。S-App を組み直す。中間の型は `eapply` がメタ変数として扱い、他方の前提を当てた時点で確定する。

E-Let。**ここが多相の場合である。**S-Let を `inversion` すると、

$$\forall \bar{\tau}.\ |\bar{\tau}| = n \Rightarrow \vdash v : \mathtt{open}\,\bar{\tau}\,\tau_0 \qquad \text{と} \qquad \emptyset, x{:}\forall^n.\tau_0 \vdash t_2 : \tau$$

が同時に得られる。代入補題が求めるのは「v が $\forall^n.\tau_0$ の全ての具体化で型付く」ことであり、$\forall^n.\tau_0 \preceq \tau'$ を `inversion` すると $\tau' = \mathtt{open}\,\bar{\tau}\,\tau_0$ で $|\bar{\tau}| = n$ という形が出るので、第一の前提をその $\bar{\tau}$ で具体化すればそのまま渡せる。**追加の作業が要らない。**

$$\dfrac{\dfrac{\vdash \mathtt{let}\ x = v\ \mathtt{in}\ t_2 : \tau}{\emptyset, x{:}\forall^n.\tau_0 \vdash t_2 : \tau}\ \text{S-Let の inversion（第二前提）} \qquad \forall \bar{\tau}.\ |\bar{\tau}| = n \Rightarrow \vdash v : \mathtt{open}\,\bar{\tau}\,\tau_0}{\vdash t_2[x := v] : \tau}\ \text{代入補題}\ \text{S-Let の inversion（第一前提）}$$

E-Beta の木と見比べると違いがはっきりする。あちらは `inversion` を二段重ね、さらに `inst_mono_inv` で全称量化を一つの型へ潰す必要があった。こちらは一段の `inversion` で二つの前提が出そろい、第一前提が既に「全ての具体化について」の形をしている。S-Let を「全ての具体化で型付く」という意味論的な形で述べたことの見返りがここにある。構文的な Gen 規則を採ると、この箇所で型代入補題と新鮮性の議論が必要になる。

E-Let1。S-Let を組み直す。第一の前提は、古い前提を各 $\bar{\tau}$ で具体化し帰納法の仮定を通して作る。本体の前提は τ_0 も n も変わらないのでそのまま持ち越す。

八つの場合が閉じる。合同規則の六つは規則を組み直すだけで、実質的な内容は E-Beta と E-Let の二つ、いずれも代入補題への帰着である。 □

8.6 実例: 多相性が実際に使われる

$id = \lambda x.\, x$ が $\forall^1.\, \#0 \to \#0$ の全ての具体化で型付くことを示す。任意の $\bar{\tau} = \tau$ について $\mathtt{open}\,[\tau]\,(\#0 \to \#0) = \tau \to \tau$ であり、S-Lam に続いて `inst_mono` つきの S-Var を使えば $\vdash \lambda x.\, x : \tau \to \tau$ が一般に導ける。この前提が S-Let を養い、

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

を与える。二つの出現はそれぞれ $[\mathtt{int} \to \mathtt{int}]$ と $[\mathtt{int}]$ で具体化される。前の版の単相体系ではこれは型付かない。最後に E-Let で一歩進め、`preservation` で型を付け直している。

9　一般化の二つの見方

二つのファイルは一般化を意図的に別の角度から扱っている。

	`HMSoundness.v`	`HMTypeSafety.v`
形	構文的	意味論的
条件	$\bar{\alpha} \cap \mathrm{ftv}(\Gamma) = \emptyset$	全ての具体化で型付く
量化変数の束縛	名前付き	de Bruijn 指標
環境	連想リスト	関数
$\mathrm{ftv}(\Gamma)$ の要否	必要	不要

意味論的な形は、型代入補題と新鮮性の機構なしに安全性の証明を閉じるための選択である。標準的な定式化ではあるが、構文的な Gen 規則そのものではない。その規則を求める読者は `HMSoundness.v` を見られたい。両者を繋ぐこと、すなわち構文的な一般化が意味論的な条件を満たすことの証明は、自然な次の一歩であり、ここでは行っていない。

10　限界の突破: 再帰、多段の安全性、occurs check

前節までの体系には、繰り返し「対象外」と記してきた欠落があった。本節はそのうち三つを埋める。いずれも Coq で証明済みで、追加公理に依存しない。

10.1　再帰を入れる

MiniML の中核は `let rec` であるのに、型安全性の証明に再帰が無かった。再帰関数を項に加える。

$$t \ ::= \ \cdots \mid \texttt{fix}\ f\ x.t$$

$\texttt{fix}\ f\ x.t$ は値であり、自分自身の名前 f が本体で使える。

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

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

S-Fix は、これから作る矢印型で f を単相に束縛したうえで本体を型付ける。E-FixApp は引数を代入し、続いて再帰を一段ほどく。

■何が変わったか。 S-Fix が増えたことで、既存の証明すべてに新しい場合が生じた。

標準形の補題。関数型の値はもはやラムダとは限らない。結論を選言に変える。

$$\mathrm{value}(v), \vdash v : \tau_1 \to \tau_2 \implies (\exists x, t.\ v = \lambda x.t) \ \vee \ (\exists f, x, t.\ v = \texttt{fix}\ f\ x.t)$$

progress。適用の場合がこの選言で分岐する。関数がラムダなら E-Beta、再帰関数なら E-FixApp が発火する。証人が二通りになっただけで、詰まらないことは変わらない。

代入補題。 `fix` $f\ x.t$ は束縛を**二つ**持つので、代入する変数がどちらに一致するかで四つの場合に分かれる。f と一致、x と一致、両方と一致、どちらとも異なる、である。前三者では代入が止まり、環境の付け替えを `context_invariance` で行う。最後の場合だけ代入が本体に入り、帰納法の仮定を使う。

preservation。 E-FixApp の場合が新しい。代入補題を**二度**使う。まず外側の束縛 x を剥がして引数 v を代入し、次に残った束縛 f を剥がして再帰関数自身を代入する。環境 $(\Gamma, f{:}\tau_1 \to \tau_2), x{:}\tau_1$ の x が外側にあるので、この順序なら束縛を順に剥がすだけで済み、f と x の相異を仮定する必要がない。

$$\dfrac{\dfrac{(\emptyset, f{:}\tau_1 \to \tau_2), x{:}\tau_1 \vdash t : \tau_2 \qquad \vdash v : \tau_1}{\emptyset, f{:}\tau_1 \to \tau_2 \vdash t[x := v] : \tau_2}\ \text{代入補題（}x\text{ を剥がす）} \qquad \vdash \texttt{fix}\ f\ x.t : \tau_1 \to \tau_2}{\vdash \big(t[x := v]\big)[f := \texttt{fix}\ f\ x.t] : \tau_2}\ \text{代入補題（}f\text{ を剥がす）}$$

10.2 多段の安全性

単段の preservation は「一歩進んでも型が保たれる」としか言わない。本来主張したいのは「型の付いた閉じた項は、何歩進んでも**詰まった状態**に到達しない」である。

$$\mathrm{stuck}(t) \ \equiv\ \neg\,\mathrm{value}(t)\ \wedge\ \neg\,\exists t'.\ t \longrightarrow t'$$

定理 6 (`soundness`)**.** $\vdash t : \tau$ かつ $t \longrightarrow^* t'$ ならば $\neg\,\mathrm{stuck}(t')$。

Proof. まず多段版の preservation を、$t \longrightarrow^* t'$ の導出に関する帰納法で示す。反射の場合は仮定そのもの、推移の場合は単段の preservation を一度使ってから帰納法の仮定に渡す。これで $\vdash t' : \tau$ が得られる。あとは t' に progress を適用すれば、t' は値であるか一歩進むかのいずれかであり、どちらも $\mathrm{stuck}(t')$ の二つの連言肢と矛盾する。 □

これが「型の付いたプログラムは誤らない」の完全な形である。

■停止性は主張していない。 再帰が入った以上、型が付いても止まらない項が書ける。実際 `fix` $f\ x.f\,x$ は任意の $\tau_1 \to \tau_2$ に型が付き、$(\texttt{fix}\ f\ x.f\,x)\,0$ は自分自身へ簡約される。この項について `loop_never_stuck` を証明してある。永遠に走り続けることと詰まることは別であり、型安全性が保証するのは後者が起きないことだけである。この区別がはっきり見えるようになったのは、再帰を入れたからである。

10.3 occurs check

単一化は、変数 α がその型の内部に真に現れるとき、方程式 $\alpha = \tau$ を拒否しなければならない。これは約束事ではなく定理である。

定理 7 (`occurs_check_sound`)**.** α が $\tau_1 \to \tau_2$ に出現するならば、$s(\alpha) = s(\tau_1 \to \tau_2)$ を満たす代入 s は存在しない。

Proof. 型の大きさ size を、`int` と変数を 1、$\mathrm{size}(\tau_1 \to \tau_2) = 1 + \mathrm{size}(\tau_1) + \mathrm{size}(\tau_2)$ と定める。

補題を二つ用意する。第一に、α が τ のどこかに現れるなら、任意の s について $\mathrm{size}(s(\alpha)) \leq \mathrm{size}(s(\tau))$。これは τ に関する帰納法で、矢印の場合は出現がどちらの枝にあるかで分ける。第二に、α が矢印型に現れるなら不等号は**狭義**になる。矢印の size が両枝の和より 1 大きく、どの型の size も 1 以上だからである。

さて $s(\alpha) = s(\tau_1 \to \tau_2)$ を満たす s があったとする。第二の補題は $\mathrm{size}(s(\alpha)) < \mathrm{size}(s(\tau_1 \to \tau_2))$ を与えるが、仮定でこの二つは同じ項なので $n < n$ となり矛盾する。 □

系 8. $\alpha = \alpha \to \alpha$ は解を持たない。

これはちょうど $\lambda x.\, x\, x$ を素朴に単一化したときに現れる方程式である。occurs check は、そこでアルゴリズムが停止せずに無限に型を作り続けるのを止めている。

11　なお残る範囲と限界

これは MiniML 実装の検証では**ない**。証明されたのは上の二つの体系についてであって、このリポジトリの SML コードとの隔たりは大きい。次は未着手である。

- 単一化アルゴリズムそのもの。occurs check の必要性は第 10 節で証明したが、単一化の手続きを定義してその健全性と完全性を示してはいない。I-Lam は依然として τ_1 を推測し、I-App は関数が既に矢印型であることを前提にする。
- principal type 定理。
- 構文的な一般化と意味論的な一般化の橋渡し。
- リスト、組、パターン照合。
- SML 実装との対応証明。

再帰と多段の安全性は第 10 節で埋めた。以前この一覧にあった「再帰」と「occurs check」は、そこへ移っている。

11.1　証明が覆っていない反例

MiniML は比較演算子に $\forall \alpha.\, \alpha \times \alpha \to \texttt{bool}$ という型を与えているが、関数値は実行時に比較できない。次は型検査を通ってから失敗する。

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

これは現状の MiniML に対する progress の反例である。どちらの Coq ファイルにも比較演算子は無いので、どちらもこれを排除しない。各ファイルが記述している言語の範囲では、定理は成立している。MiniML を直すには、Standard ML が `''a` で行っているような等価型を入れるか、比較を単相に戻すことになる。

12 検証方法

Rocq Prover 9.1.0 で確認した。

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

どちらも終了コード 0 で成功し、`From Coq` が `From Stdlib` に置き換わった旨の deprecation 警告以外に出力は無い。`Admitted` も `admit` も `Axiom` も無く、38 個の証明はすべて `Qed` で閉じている。さらに主要な定理について隠れた仮定の有無を監査した。

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

いずれも `Closed under the global context` と答えるので、追加公理に依存していない。

本稿は次で組んだ。

```
lualatex -interaction=nonstopmode hm_safety_report_ja.tex
```