AIPL における型の健全性と型の安全性

—- 証明が成立する範囲 AIPL$^-$ の定式化と機械検証 —-

2026 年 7 月

概要

AIPL（ABCL/c+）は ABCL 系の actor-first 並行オブジェクト言語であり、Hindley–Milner 風の型推論、actor メソッド表、future/await、実行時セッションプロトコル検査を備える。しかし現行実装ではメソッドの返り値型が抽象構文木に存在せず、`reply` と `now` / `future` の戻り値が型で結ばれていないため、これらは `any` に落ちている。本レポートは、この一点だけを直した部分言語 **AIPL$^-$** を定義し、その型健全性（保存・進行）と型安全性（well-typed な構成から到達できる構成は行き詰まらない）を Coq で機械検証した記録である。形式化は変数・局所束縛・代入補題、actor の状態 store とその型不変条件、クラス表とメソッド表、メールボックスと非同期送信、future 表・`await`・動的 actor 生成をすべて含む。主定理は六つで、いずれも公理に依存しない。進行定理は **値・簡約・***await* **待ち**の三択として述べる —- デッドロックは型だけでは排除できないからであり、その境界を明示することが本レポートの主眼の一つである。その上で、**哲学者の食事問題**を AIPL$^-$ で書き、そのプログラムがデッドロックしないことを二層に分けて証明する。第 1 層は機械的デッドロック自由（プログラムが `await` を使わないこと）、第 2 層は資源デッドロック自由（フォークを番号順に取ること）であり、後者は**優先度つきセッション型**の健全性証明にほかならない。優先度規律が必要でもあることを、素朴な割り当てに対する詰まり状態の構成によって示す。

目次

1 位置づけと、先にお断りすること

1.1 対象

本レポートの対象は `yaskodama/abclcp` の実装（OCaml。parser、型推論、actor runtime、future/await、AIOS primitive、session protocol runtime からなる）である。以下のファイルを読んで体系を抽出した。

ファイル	内容
`src/ast.ml`	抽象構文木。式・文・class・method・send target・select case
`src/parser.mly`, `src/lexer.mll`	構文
`src/types.ml`	型、型スキーム、単一化、class method registry
`src/infer.ml`	型推論・型検査
`src/eval_thread.ml`	actor runtime、mailbox、future、reply、protocol

先行する社内レポート `docs/ABCLCP_TYPE_SOUNDNESS_REPORT.md` は、実装の型システムを整理し、Coq 形式化 `coq/AbclSoundness.v` を与えている。ただしその形式化は**閉じた式**の計算体系（`nat`, `string`, `unit`, `future`, `+`, `await`）であって、変数も代入も actor も mailbox も含まない。同レポート第 9 節は、拡張の順序として次を挙げていた。

1. 変数、環境、代入、substitution lemma を追加する。
2. 関数型と primitive overload の soundness を追加する。
3. actor store と method table を追加する。
4. mailbox と send の small-step semantics を追加する。
5. future table と reply の対応を追加する。
6. session environment を静的型判断へ移す。
7. `any` と remote boundary を gradual typing として定式化する。

本レポートは 1, 3, 4, 5 を実行したものである。2（overload と多相）、6（静的セッション型）、7（`any`）は範囲外とした。理由は §3 に書く。

1.2 AIPL$^-$ は AIPL そのものではない

本レポートが証明するのは AIPL そのものではなく、**AIPL$^-$**（AIPL マイナーバージョン）である。AIPL$^-$ は AIPL から機能を削っただけでなく、一箇所だけ**言語を変更**している。

> **変更点。**メソッドに返り値型を宣言させ、`reply(v)` を「メソッド本体の値」とする。

現行 AIPL では `method m(x1,...,xn)` に返り値型注釈が無く、型検査器は `reply` の値とメソッ

ドの戻り値を結びつけていない。そのため `now` は `any`、`future` は `future any` に落ちる。§3.2 で述べるとおり、この一点を直すと future/await を含めた完全な型健全性が証明できるようになる。逆に言えば、**現行 AIPL のままでは型安全性は成立しない**。これが本レポートの実装へのフィードバックである。

2 現行 AIPL の型システムが保証していること

証明の前に、実装の現在地を確認しておく。

2.1 型

`src/types.ml` の型は次である。

$$\begin{aligned}\tau ::= \alpha \mid \texttt{int} \mid \texttt{float} \mid \texttt{string} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{any} \mid \tau[] \mid \texttt{future}\ \tau \\ \mid (\tau_1 \times \cdots \times \tau_n) \to \tau \mid \texttt{actor}(C)\{m_1{:}\tau_1; \ldots; m_n{:}\tau_n\} \mid \{l_1{:}\tau_1; \ldots; l_n{:}\tau_n\}\end{aligned}$$

型変数は mutable link による union-find で表現され、`repr` が経路圧縮を行う。型スキームは $\sigma ::= \forall \alpha_1 \ldots \alpha_n.\ \tau$、型環境は名前からスキームの**リスト**への写像である（primitive overload のため）。

2.2 保証していること

- 基底型・配列・future・actor の基本的な型整合性。
- primitive 呼び出しの arity と overload 一致。
- ローカル actor への `send` / `now` / `future` で、target が actor であり、メソッドが存在し、引数の個数と型が合うこと。
- `await` の対象が `future` τ であること。
- `if` / `while` の条件が `bool` であること。
- `become` の `init` 引数が合うこと。

2.3 保証していないこと

- `reply` の値型と `now` / `future` の戻り値型の一致**（これが本レポートで直す点）**。
- remote actor のメソッド存在、`sender` の actor 型、`send!`（unsafe send）の target/method 整合性。
- `select` パターンと mailbox プロトコルの静的整合性。
- `become` 前後のプロトコル互換性。
- セッションプロトコルの静的遵守（現在は runtime 検査）。
- デッドロック自由。

注意 1. これらは欠陥というより、実装が「実用的な actor runtime + 漸進的な型システム」の段階にあることを意味する。証明の対象を切るときは、この境界に沿って切るのが正しい。

3 AIPL⁻ —- 証明が成立する範囲

3.1 残したもの・落としたもの

AIPL の要素	AIPL⁻	理由
変数、局所変数束縛 `var x = e`	残す	代入補題の対象
算術・比較・条件分岐	残す	基本
actor の状態（フィールド）	残す（1 個）	store の型不変条件を言うため
クラス表・メソッド表	残す	メソッド解決の核
`new C` による actor 生成	残す	表が動的に伸びる。単調性が要る
非同期送信（past）	残す	糖衣として表現（§3.3）
`future` 送信・`await`	残す	本レポートの中心
`reply`	**変更**	本体の値とする（§3.2）
`self`	残す	状態アクセスとクラス文脈
`any`	落とす	動的境界。健全性の定義そのものを弱める
`send!`（unsafe send）	落とす	明示的な escape hatch
remote actor, `sender`	落とす	動的境界
`become`	落とす	切替後のプロトコル互換性が未解決
`select`, `timeout`	落とす	スケジューリングであり型ではない
配列・レコード	落とす	直交する。証明を長くするだけ
overload・型スキーム（多相）	落とす	単一化アルゴリズムの健全性が別課題
`while`	落とす	停止性を論じないので `if` で足りる
セッションプロトコル	落とす	静的型に昇格していない

落とした項目のうち `any` は決定的である。`any` があると「well-typed programs cannot go wrong」がそのままでは主張できず、「静的型付き断片の中では詰まらない。動的境界では失敗しうるが、失敗は境界で明示的」という弱い形にしかならない。AIPL⁻ は `any` を持たないので、無条件の主張ができる。

3.2 唯一の言語変更 —- reply と返り値型

現行 AIPL のメソッド宣言は

```
class Counter {
  var count = 0;
  method inc() {
    count = count + 1;
    reply(count); // 型検査器はこの値を追跡しない
  }
}
var c = new Counter();
var x = now c.inc(); // x : any
```

であり、型検査器はメソッド型を $(\alpha_1 \times \cdots \times \alpha_n) \to \texttt{unit}$ として登録する。`reply` は runtime

では future を解決するが、型の上では戻り値と無関係である。結果として `now` は `any`、`future` は `future any` になる。

AIPL$^-$ では次のように書く。

```
class Counter : int { // 状態の型を宣言
  method inc(x : unit) : int { // 引数型と返り値型を宣言
    set(get() + 1);
    get() // 本体の値がそのまま reply の値
  }
}
var c = new Counter();
var x = await (future c.inc(())); // x : int
```

すなわち：

1. メソッド表 $\mathrm{mtab}(c, m) = (\tau_a, \tau_r)$ が**引数型と返り値型の対**を持つ。
2. メソッド本体は τ_r 型でなければならない（§4.3 の BodiesOk）。
3. `reply(v)` という文は無く、本体の評価結果が future を解決する（規則 CFinish）。

これだけで、mtab の τ_r、future 表 Φ の型、そして `await` の結果型が一本の鎖でつながる。定理 13 がその帰結である。

3.3 糖衣

AIPL$^-$ の式は少ないが、AIPL の送信三態のうち二つは糖衣で書ける。

$$
\begin{aligned}
e_1; e_2 &\equiv \texttt{let}\ z = e_1\ \texttt{in}\ e_2 && (z \text{ は } e_2 \text{ に現れない}) \\
\texttt{send}\ t.m(e) &\equiv \texttt{let}\ z = (t \Leftarrow m(e))\ \texttt{in}\ () && \text{past 型: future を捨てる} \\
\texttt{now}\ t.m(e) &\equiv \texttt{await}\ (t \Leftarrow m(e)) && \text{同期: 直ちに待つ}
\end{aligned}
$$

したがって規則を増やす必要はない。past 型送信が「future を作って捨てる」というのは実装の実際（`now` は future send を生成して即 `await` する）とも一致している。

4 構文

4.1 型

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{actor}[c] \mid \texttt{future}\ \tau \qquad (c \in \mathbb{N})$$

$\texttt{actor}[c]$ の c はクラス番号であり、メソッド表への索引である。オブジェクトの型は構造でもインタフェース列でもなく、**どのメソッド表を引くか**を指す。

4.2 式

$e ::= n \mid true \mid false \mid ()$	リテラル
$\mid x$	変数
$\mid$ `self`	自分自身
$\mid o \mid \kappa$	actor 参照 / future 参照（実行時のみ）
$\mid e_1 + e_2 \mid e_1 < e_2$	
$\mid$ `if` e_0 `then` e_1 `else` e_2	
$\mid$ `let` $x = e_1$ `in` e_2	`var x = e1; e2`
$\mid$ `get` $\mid$ `set` e	自分の状態の読み書き
$\mid$ `new` c	actor 生成
$\mid e_0 \Leftarrow m(e_1)$	future 送信
$\mid$ `await` e	

値は $v ::= n \mid true \mid false \mid () \mid o \mid \kappa$ である。o と κ は実行時にのみ現れる。
代入 $e[x := v]$ は素直な再帰で定義するが、`let` の束縛が x を隠す場合は本体に入らない。

$$(\texttt{let } y = e_1 \texttt{ in } e_2)[x := v] = \texttt{let } y = e_1[x := v] \texttt{ in } \begin{cases} e_2 & (y = x) \\ e_2[x := v] & (y \neq x) \end{cases}$$

代入される値 v は閉じている（型付けが空環境でつく）ので、変数捕獲は起こらない。

4.3 プログラム

プログラムは四つの表で与えられる。Coq では `Section` の `Variable` にしてあるので、定理を閉じたときには全称量化された引数になる。

$\mathrm{stype} : \mathbb{N} \to \tau$	クラス c の状態の型
$\mathrm{sinit} : \mathbb{N} \to e$	クラス c の状態の初期値
$\mathrm{mtab} : \mathbb{N} \to \mathbb{N} \to (\tau \times \tau)_\bot$	$c.m$ の（引数型, 返り値型）
$\mathrm{mbody} : \mathbb{N} \to \mathbb{N} \to e$	$c.m$ の本体。引数は変数 0

そして三つの前提を置く。

定義 2 (プログラムの型検査)**.**

(SInitValue) $\forall c.\ \mathrm{sinit}(c)$ は値 (1)

(SInitOk) $\forall c, \Omega, \Phi, C, \Gamma.\ \Omega; \Phi; C; \Gamma \vdash \mathrm{sinit}(c) : \mathrm{stype}(c)$ (2)

(BodiesOk) $\forall c, m, \tau_a, \tau_r.\ \mathrm{mtab}(c, m) = (\tau_a, \tau_r) \implies \forall \Omega, \Phi.\ \Omega; \Phi; c; x_0{:}\tau_a \vdash \mathrm{mbody}(c, m) : \tau_r$ (3)

BodiesOk が「プログラム全体が型検査を通っている」ことである。$\forall \Omega, \Phi$ となっているのは、メソッド本体はソースコードであり実行時参照 o, κ を含まないので、どの表のもとでも同じ型が付くからである。

注意 3. これらは Coq の `Axiom` ではなく `Section` の `Hypothesis` である。`End` の時点で各定理の**前提**に変わるので、`Print Assumptions` は「Closed under the global context」を返す。公理を追加してはいない。

5 型付け規則

型判断は

$$\Omega; \Phi; C; \Gamma \vdash e : \tau$$

の形である。Ω はオブジェクト表（actor 番号 $\mapsto$ クラス番号）、Φ は future 表（future 番号 $\mapsto$ その型）、C は**いまどのクラスの本体を型検査しているか**、Γ は変数の型環境である。

$$\frac{}{\Omega; \Phi; C; \Gamma \vdash n : \texttt{int}}\ \textsc{HNum} \qquad \frac{}{\Omega; \Phi; C; \Gamma \vdash b : \texttt{bool}}\ \textsc{HBool} \qquad \frac{}{\Omega; \Phi; C; \Gamma \vdash () : \texttt{unit}}\ \textsc{HUnit}$$

$$\frac{\Gamma(x) = \tau}{\Omega; \Phi; C; \Gamma \vdash x : \tau}\ \textsc{HVar} \qquad \frac{}{\Omega; \Phi; C; \Gamma \vdash \texttt{self} : \texttt{actor}[C]}\ \textsc{HSelf}$$

$$\frac{\Omega(o) = c}{\Omega; \Phi; C; \Gamma \vdash o : \texttt{actor}[c]}\ \textsc{HORef} \qquad \frac{\Phi(\kappa) = \tau}{\Omega; \Phi; C; \Gamma \vdash \kappa : \texttt{future}\ \tau}\ \textsc{HFRef}$$

$$\frac{\Omega; \Phi; C; \Gamma \vdash a : \texttt{int} \qquad \Omega; \Phi; C; \Gamma \vdash b : \texttt{int}}{\Omega; \Phi; C; \Gamma \vdash a + b : \texttt{int}}\ \textsc{HAdd}$$

$$\frac{\Omega; \Phi; C; \Gamma \vdash a : \texttt{int} \qquad \Omega; \Phi; C; \Gamma \vdash b : \texttt{int}}{\Omega; \Phi; C; \Gamma \vdash a < b : \texttt{bool}}\ \textsc{HLt}$$

$$\frac{\Omega; \Phi; C; \Gamma \vdash a : \texttt{bool} \qquad \Omega; \Phi; C; \Gamma \vdash b : \tau \qquad \Omega; \Phi; C; \Gamma \vdash d : \tau}{\Omega; \Phi; C; \Gamma \vdash \texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ d : \tau}\ \textsc{HIf}$$

$$\frac{\Omega; \Phi; C; \Gamma \vdash e_1 : \tau_1 \qquad \Omega; \Phi; C; \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Omega; \Phi; C; \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \textsc{HLet} \qquad \frac{}{\Omega; \Phi; C; \Gamma \vdash \texttt{get} : \mathrm{stype}(C)}\ \textsc{HGet}$$

$$\frac{\Omega; \Phi; C; \Gamma \vdash e : \mathrm{stype}(C)}{\Omega; \Phi; C; \Gamma \vdash \texttt{set}\ e : \texttt{unit}}\ \textsc{HSet} \qquad \frac{}{\Omega; \Phi; C; \Gamma \vdash \texttt{new}\ c : \texttt{actor}[c]}\ \textsc{HNew}$$

$$\frac{\Omega; \Phi; C; \Gamma \vdash e_0 : \texttt{actor}[c] \qquad \mathrm{mtab}(c, m) = (\tau_a, \tau_r) \qquad \Omega; \Phi; C; \Gamma \vdash e_1 : \tau_a}{\Omega; \Phi; C; \Gamma \vdash e_0 \Leftarrow m(e_1) : \texttt{future}\ \tau_r}\ \textsc{HSend}$$

$$\frac{\Omega; \Phi; C; \Gamma \vdash e : \texttt{future}\ \tau}{\Omega; \Phi; C; \Gamma \vdash \texttt{await}\ e : \tau}\ \textsc{HAwait}$$

読みどころは三つある。

■HSelf と HGet / HSet が C を使う。 `self` の型は $\texttt{actor}[C]$、`get` の型は $\mathrm{stype}(C)$ である。つまり**同じ式が、どのクラスの本体に置かれるかで型が変わる**。これが型判断にクラス文脈 C を持たせている理由である。

■HSend が mtab を引く。 送信先の型 $\texttt{actor}[c]$ から c を読み出し、$\mathrm{mtab}(c, m)$ が定義されていなければ型が付かない。**これが** *message not understood* **を静的に潰している箇所**である。そして結論の型が $\texttt{future}\ \tau_r$ —- τ_r は mtab が持つ返り値型である。現行 AIPL ではここが $\texttt{future any}$ になっていた。

■HORef, HFRef が実行時の表を引く。 型判断が Ω, Φ に依存するのはこの二規則のためだけである。両表は実行中に**伸びるだけ**なので、補題 15（単調性）が成り立つ。

6 操作的意味論

6.1 実行時の構造

$H = (\Omega, \Sigma, \Phi, \Psi)$	ヒープ
Ω : actor 番号 $\to$ クラス番号	追記のみ
Σ : actor 番号 $\to$ 状態値	追記と一点更新
Φ : future 番号 $\to \tau$	追記のみ
Ψ : future 番号 $\to e_\perp$	追記と一点更新

$M = \langle o \Leftarrow m(v) \rhd \kappa \rangle$	メッセージ
$t = [\, o \vdash e \rhd \kappa \,]$	タスク: actor o で走り、終われば κ を解決
$\mathcal{C} = (H;\ \overline{M};\ \bar{t}\,)$	構成

注意 4 (store typing が要らない理由). 一般に可変 store を持つ体系では store typing Σ を型判断に追加し、$\Sigma \subseteq \Sigma'$ の下での弱化を証明する。AIPL$^-$ では **actor** o **の状態の型は** $\mathrm{stype}(\Omega(o))$ **で決まる**ので、別の store typing が要らない。伸びるのは Ω と Φ だけであり、それらの単調性（補題 15）だけで済む。これは「状態の型をクラス宣言に固定する」という言語設計が証明を軽くしている例である。

6.2 タスク一歩

$H; o \vdash e \ \longrightarrow\ H'; \mu; e'$ と書く。o は自分自身、μ は送出されるメッセージ列である。

■基底規則。

$$\frac{}{H; o \vdash n + k \;\longrightarrow\; H; \epsilon; n{+}k}\ \textsc{StAdd} \qquad \frac{}{H; o \vdash n < k \;\longrightarrow\; H; \epsilon; n < k}\ \textsc{StLt}$$

$$\frac{}{H; o \vdash \texttt{if}\ true\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \;\longrightarrow\; H; \epsilon; e_1}\ \textsc{StIfT}$$

$$\frac{}{H; o \vdash \texttt{if}\ false\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \;\longrightarrow\; H; \epsilon; e_2}\ \textsc{StIfF}$$

$$\frac{v \text{ は値}}{H; o \vdash \texttt{let}\ x = v\ \texttt{in}\ e \;\longrightarrow\; H; \epsilon; e[x := v]}\ \textsc{StLet} \qquad \frac{}{H; o \vdash \texttt{self} \;\longrightarrow\; H; \epsilon; o}\ \textsc{StSelf}$$

$$\frac{\Sigma(o) = v}{H; o \vdash \texttt{get} \;\longrightarrow\; H; \epsilon; v}\ \textsc{StGet} \qquad \frac{v \text{ は値}}{H; o \vdash \texttt{set}\ v \;\longrightarrow\; (\Omega, \Sigma[o \mapsto v], \Phi, \Psi); \epsilon; ()}\ \textsc{StSet}$$

$$\frac{}{H; o \vdash \texttt{new}\ c \;\longrightarrow\; (\Omega \cdot c,\ \Sigma \cdot \mathrm{sinit}(c),\ \Phi,\ \Psi); \epsilon; |\Omega|}\ \textsc{StNew}$$

$$\frac{v \text{ は値} \qquad \Omega(o') = c \qquad \mathrm{mtab}(c, m) = (\tau_a, \tau_r)}{H; o \vdash o' \Leftarrow m(v) \;\longrightarrow\; (\Omega, \Sigma, \Phi \cdot \tau_r, \Psi \cdot \bot); \langle o' \Leftarrow m(v) \rhd |\Phi| \rangle; |\Phi|}\ \textsc{StSend}$$

$$\frac{\Psi(\kappa) = v}{H; o \vdash \texttt{await}\ \kappa \;\longrightarrow\; H; \epsilon; v}\ \textsc{StAwait}$$

■合同規則。 部分式が進めば全体も進む。`add`, `lt`, `if`（条件のみ）、`let`（束縛式のみ）、`set`, `send`（左右）、`await` の 10 規則。形はすべて同じなので、代表して二つだけ書く。

$$\frac{H; o \vdash a \;\longrightarrow\; H'; \mu; a'}{H; o \vdash a \Leftarrow m(b) \;\longrightarrow\; H'; \mu; a' \Leftarrow m(b)}\ \textsc{StSend1}$$

$$\frac{v \text{ は値} \qquad H; o \vdash b \;\longrightarrow\; H'; \mu; b'}{H; o \vdash v \Leftarrow m(b) \;\longrightarrow\; H'; \mu; v \Leftarrow m(b')}\ \textsc{StSend2}$$

6.3 await 待ち

未解決の future を待って止まっている状態を、評価文脈に沿った述語として定義する。

$$\frac{\Psi(\kappa) = \bot}{\mathrm{awaiting}(H, \texttt{await}\ \kappa)}\ \textsc{AwHere} \qquad \frac{v \text{ は値} \qquad \mathrm{awaiting}(H, b)}{\mathrm{awaiting}(H, v \Leftarrow m(b))}\ \textsc{AwSend2}$$

$$\frac{\mathrm{awaiting}(H, a)}{\mathrm{awaiting}(H, \texttt{await}\ a)}\ \textsc{AwAwait}$$

（ほか `add` 左右、`lt` 左右、`if`、`let`、`set`、`send` 左の 8 規則。合同規則と 1 対 1 に対応する。）

6.4 構成一歩

$$\frac{H; o \vdash e \;\longrightarrow\; H'; \mu; e'}{(H;\ \overline{M};\ \overline{t_1}, [\, o \vdash e \rhd \kappa \,], \overline{t_2}) \Longrightarrow (H';\ \overline{M} \cdot \mu;\ \overline{t_1}, [\, o \vdash e' \rhd \kappa \,], \overline{t_2})}\ \textsc{CTask}$$

$$\frac{v \text{ は値}}{(H;\ \overline{M};\ \overline{t_1}, [\, o \vdash v \rhd \kappa \,], \overline{t_2}) \Longrightarrow ((\Omega, \Sigma, \Phi, \Psi[\kappa \mapsto v]);\ \overline{M};\ \overline{t_1}, \overline{t_2})}\ \textsc{CFinish}$$

$$\frac{\Omega(o) = c \qquad \mathrm{mtab}(c, m) = (\tau_a, \tau_r)}{(H;\ \overline{M_1}, \langle o \Leftarrow m(v) \rhd \kappa \rangle, \overline{M_2};\ \bar{t}\,) \Longrightarrow (H;\ \overline{M_1}, \overline{M_2};\ \bar{t}, [\, o \vdash \mathrm{mbody}(c, m)[x_0 := v] \rhd \kappa \,])}\ \textsc{CDeliver}$$

CFinish が reply である。タスクの式が値になったとき、その値が future κ を解決する。別に `reply` 構文を持たない。これが §3.2 の言語変更の操作的な姿である。

CDeliver が actor 起動である。メッセージを取り出し、宛先のクラスを引き、そのメソッド本体に引数を代入して新しいタスクにする。返すべき future κ はメッセージが運んでいる。

7 不変条件

定義 5 (ヒープの正しさ)**.** heap_ok(H) とは次の四つ：

1. $|\Sigma| = |\Omega|$（すべての actor が状態を持つ）
2. $|\Psi| = |\Phi|$（すべての future が枠を持つ）
3. $\Omega(o) = c \wedge \Sigma(o) = v \implies v$ は値 $\wedge\, \forall C, \Gamma.\ \Omega; \Phi; C; \Gamma \vdash v : \mathrm{stype}(c)$
4. $\Phi(\kappa) = \tau \wedge \Psi(\kappa) = v \implies v$ は値 $\wedge\, \forall C, \Gamma.\ \Omega; \Phi; C; \Gamma \vdash v : \tau$

3 が**状態の型不変条件**、4 が **future の型不変条件**である。「$\forall C, \Gamma$」となっているのは、値は actor 間をメッセージで移動するため、特定のクラス文脈に縛られていてはいけないからである（補題 16)。

定義 6 (メッセージ・タスク・構成の正しさ)**.**

$$\begin{aligned}
\mathrm{msg_ok}(H, \langle o \Leftarrow m(v) \rhd \kappa \rangle) \iff{}& \exists c, \tau_a, \tau_r.\ \Omega(o) = c \wedge \mathrm{mtab}(c, m) = (\tau_a, \tau_r) \\
&\wedge\ v \text{ は値} \wedge (\forall C, \Gamma.\ \Omega; \Phi; C; \Gamma \vdash v : \tau_a) \wedge \Phi(\kappa) = \tau_r \\
\mathrm{task_ok}(H, [\, o \vdash e \rhd \kappa \,]) \iff{}& \exists c, \tau.\ \Omega(o) = c \wedge \Phi(\kappa) = \tau \wedge \Omega; \Phi; c; \emptyset \vdash e : \tau \\
\mathrm{conf_ok}(H; \overline{M}; \bar{t}\,) \iff{}& \mathrm{heap_ok}(H) \wedge (\forall M \in \overline{M}.\ \mathrm{msg_ok}(H, M)) \wedge (\forall t \in \bar{t}.\ \mathrm{task_ok}(H, t))
\end{aligned}$$

task_ok の $\Omega; \Phi; c; \emptyset \vdash e : \tau$ の c は**タスクが走っている** *actor* **のクラス**であり、τ は**そのタスクが解決すべき** *future* **の型**である。この二つが一致していることが、`self`/`get`/`set` の健全性と、reply の健全性を同時に支える。

8 主定理

定理 7 (理解できないメッセージは飛ばない)**.** conf_ok($H; \overline{M}; \bar{t}\,$) かつ $\langle o \Leftarrow m(v) \rhd \kappa \rangle \in \overline{M}$ ならば

$$\exists c, \tau_a, \tau_r.\ \Omega(o) = c \ \wedge\ \mathrm{mtab}(c, m) = (\tau_a, \tau_r) \ \wedge\ (\forall C, \Gamma.\ \Omega; \Phi; C; \Gamma \vdash v : \tau_a) \ \wedge\ \Phi(\kappa) = \tau_r.$$

宛先の actor が実在し、そのクラスがそのメソッドを持ち、引数の型が合い、**さらに返すべき future の型がメソッドの返り値型と一致している**。最後の連言が、現行 AIPL では言えなかった部分である。

定理 8 (保存)**.** conf_ok($\mathcal{C}$) かつ $\mathcal{C} \Longrightarrow \mathcal{C}'$ ならば conf_ok($\mathcal{C}'$)。

定理 9 (進行)**.** conf_ok($\mathcal{C}$) ならば、次のいずれかが成り立つ。

1. $\mathcal{C}$ は終状態（メッセージもタスクも無い)。
2. ある $\mathcal{C}'$ が存在して $\mathcal{C} \Longrightarrow \mathcal{C}'$。
3. $\mathcal{C}$ はブロックしている。すなわちメッセージが無く、タスクは空でなく、**すべてのタスクが未解決 *future* を *await* している**。

注意 10 (なぜ三択なのか). 第三の枝がデッドロックである。`await` を持つ言語では、これを型だけで排除することはできない。たとえば自分自身に送って自分で待てば必ずブロックする。本レポートの立場は、**デッドロックを排除できないことを認め、それ以外の詰まり方が起きないことを証明する**である。第三の枝が「未解決 future の await」に限定されていること自体が主張である —- 型不一致で止まることも、存在しないメソッドで止まることも、範囲外アクセスで止まることもない。

定理 11 (型安全性)**.** $\mathrm{stuck}(\mathcal{C}) \iff \neg\mathrm{terminal}(\mathcal{C}) \wedge \neg\mathrm{blocked}(\mathcal{C}) \wedge \neg\exists\mathcal{C}'.\ \mathcal{C} \Longrightarrow \mathcal{C}'$ と定義する。このとき

$$\mathrm{conf\_ok}(\mathcal{C}) \wedge \mathcal{C} \Longrightarrow^* \mathcal{C}' \implies \neg\mathrm{stuck}(\mathcal{C}').$$

定理 12 (状態の型不変条件)**.** conf_ok($\mathcal{C}$) かつ $\mathcal{C} \Longrightarrow^* (H'; \cdot; \cdot)$ ならば、H' のどの *actor* o についても、その状態は値であり、宣言された型 $\mathrm{stype}(\Omega'(o))$ を持つ。

定理 13 (future の型不変条件 —- reply と戻り値型の一致)**.** conf_ok($\mathcal{C}$) かつ $\mathcal{C} \Longrightarrow^* (H'; \cdot; \cdot)$ ならば、H' の解決済みのどの *future* κ についても、その値は値であり、$\Phi'(\kappa)$ 型を持つ。

定理 13 が §3.2 の言語変更の見返りである。`await` κ が返す値は必ず $\Phi(\kappa)$ 型であり、$\Phi(\kappa)$ は HSend により mtab の返り値型と一致し、それは BodiesOk によりメソッド本体の型と一致する。**reply、future、await の三点が一本の鎖でつながった。**

■機械検証。

```
$ coqc AIPLSoundness.v
$ Print Assumptions no_method_not_understood. -> Closed under the global context
$ Print Assumptions preservation. -> Closed under the global context
$ Print Assumptions progress. -> Closed under the global context
$ Print Assumptions type_safety. -> Closed under the global context
$ Print Assumptions state_type_invariant. -> Closed under the global context
$ Print Assumptions future_type_invariant. -> Closed under the global context
```

Coq / Rocq Prover 9.1.0。ファイルは 982 行、補題・定理 47 個。公理も未証明の補題も使っていない。

9 証明

9.1 準備 —- 三つの構造補題

補題 14 (型環境の外延性)**.** $(\forall z.\ \Gamma_1(z) = \Gamma_2(z))$ かつ $\Omega; \Phi; C; \Gamma_1 \vdash e : \tau$ ならば $\Omega; \Phi; C; \Gamma_2 \vdash e : \tau$。

型環境を関数 $\mathbb{N} \to \tau_\perp$ で表しているため、外延的に等しい環境を入れ替える補題が要る。関数外延性の公理は**使わない**。導出に関する帰納法で示す。`let` の場合、Γ_i が $\Gamma_i, x{:}\tau_1$ に変わるが、点ごとの一致は保たれる。

補題 15 (表の単調性)**.** $\Omega \sqsubseteq \Omega'$ かつ $\Phi \sqsubseteq \Phi'$ かつ $\Omega; \Phi; C; \Gamma \vdash e : \tau$ ならば $\Omega'; \Phi'; C; \Gamma \vdash e : \tau$。ここで $\Omega \sqsubseteq \Omega'$ は $\forall n, x.\ \Omega(n) = x \implies \Omega'(n) = x$ の意である。

導出に関する帰納法。Ω, Φ を使うのは HORef, HFRef だけであり、そこで $\sqsubseteq$ をそのまま適用する。リストへの追記は $\sqsubseteq$ を満たす。

補題 16 (値の型付けは文脈に依存しない)**.** v が値で $\Omega; \Phi; C; \Gamma \vdash v : \tau$ ならば、任意の C', Γ' について $\Omega; \Phi; C'; \Gamma' \vdash v : \tau$。

値の形について場合分けする。値は n, b, $()$, o, κ の五つで、それぞれ HNum, HBool, HUnit, HORef, HFRef でしか型が付かず、どれも C にも Γ にも言及しない。

注意 17. この補題は形式的には自明だが、**意味は自明でない**。値が actor 間をメッセージで移動しても型が変わらないこと —- すなわち $\texttt{actor}[c]$ や $\texttt{future}\ \tau$ が「どこから見ても同じ型」であること —- を保証している。もし `self` が値だったら、この補題は成り立たず、体系全体が壊れる。`self` を STSelf で**直ちに** o に簡約するのはそのためである。

9.2 標準形

補題 18 (標準形)**.** v が値であるとき、

$$
\begin{array}{ll}
\Omega; \Phi; C; \Gamma \vdash v : \texttt{int} & \implies \exists n.\ v = n \\
\Omega; \Phi; C; \Gamma \vdash v : \texttt{bool} & \implies \exists b.\ v = b \\
\Omega; \Phi; C; \Gamma \vdash v : \texttt{actor}[c] & \implies \exists o.\ v = o \wedge \Omega(o) = c \\
\Omega; \Phi; C; \Gamma \vdash v : \texttt{future}\ \tau & \implies \exists \kappa.\ v = \kappa \wedge \Phi(\kappa) = \tau
\end{array}
$$

値の形と型付けを二重に反転すれば、五つの候補のうち四つが矛盾で消える。第三・第四の主張では、消え残った枝の前提としてそのまま $\Omega(o) = c$、$\Phi(\kappa) = \tau$ が**取り出せる**ことが重要である。これらは後で遷移の側条件として使う。

9.3 代入補題

補題 19 (代入)**.** $\Omega; \Phi; C; \Gamma, x{:}\tau_1 \vdash e : \tau$ かつ v が値で $(\forall C', \Gamma'.\ \Omega; \Phi; C'; \Gamma' \vdash v : \tau_1)$ ならば $\Omega; \Phi; C; \Gamma \vdash e[x := v] : \tau$。

Proof. e の構造に関する帰納法。反転補題で前提を取り出す。ほとんどの場合は機械的だが、二つだけ本質的な場所がある。

■変数の場合。 $e = y$。$y = x$ なら $e[x := v] = v$ であり、反転により $\tau = \tau_1$。仮定がそのまま結論になる。$y \neq x$ なら $e[x := v] = y$ であり、$\Gamma, x{:}\tau_1$ での y の型は Γ での y の型と等しい。

$$\dfrac{\dfrac{(\Gamma, x{:}\tau_1)(x) = \tau_1}{\Omega;\Phi;C;\Gamma, x{:}\tau_1 \vdash x : \tau_1}\,\textsc{HVar} \qquad x[x := v] = v \qquad \Omega;\Phi;C;\Gamma \vdash v : \tau_1}{\Omega;\Phi;C;\Gamma \vdash x[x := v] : \tau_1}\,\text{代入（変数, } y = x\text{）}$$

■`let` の場合 —- 遮蔽。 $e = (\texttt{let}\ y = e_1\ \texttt{in}\ e_2)$。$y = x$ のとき、代入は e_2 に入らない。このとき型付けの前提は $\Omega;\Phi;C;(\Gamma, x{:}\tau_1), x{:}\tau' \vdash e_2 : \tau_2$ であり、示すべきは $\Omega;\Phi;C;\Gamma, x{:}\tau' \vdash e_2 : \tau_2$ である。この二つの環境は**外延的に等しい**：x ではどちらも τ'、$z \neq x$ ではどちらも $\Gamma(z)$。よって補題 14 が使える。

$$\dfrac{\dfrac{\dfrac{\Omega;\Phi;C;\Gamma, x{:}\tau_1 \vdash e_1 : \tau'}{\Omega;\Phi;C;\Gamma \vdash e_1[x := v] : \tau'}\,\text{IH}_1 \qquad \dfrac{\Omega;\Phi;C;(\Gamma, x{:}\tau_1), x{:}\tau' \vdash e_2 : \tau_2 \qquad (\Gamma, x{:}\tau_1), x{:}\tau' \equiv \Gamma, x{:}\tau'}{\Omega;\Phi;C;\Gamma, x{:}\tau' \vdash e_2 : \tau_2}\,\text{補題 14}}{\Omega;\Phi;C;\Gamma \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)[x := v] : \tau_2}\,\textsc{HLet}$$

$y \neq x$ のときは代入が e_2 に入る。帰納法の仮定を使うために $(\Gamma, x{:}\tau_1), y{:}\tau' \equiv (\Gamma, y{:}\tau'), x{:}\tau_1$ を示す必要があり、これも補題 14 である（$z = y$ なら両辺 τ'、$z = x \neq y$ なら両辺 τ_1、その他は $\Gamma(z)$）。 □

9.4 局所進行性

補題 20 (局所進行性)**.** heap_ok(H)、$\Omega(o) = c$、$\Omega;\Phi;c;\Gamma \vdash e : \tau$、$\Gamma$ が空（すべての変数について未定義）ならば、次のいずれか：

1. e は値。
2. ある H', μ, e' が存在して $H; o \vdash e \longrightarrow H'; \mu; e'$。
3. awaiting(H, e)。

Proof. 型付け導出に関する帰納法。Γ が空なので HVar の場合は $\Gamma(x) = \tau$ と $\Gamma(x) = \bot$ の矛盾で消える。

基底 n, b, (), o, κ は値。`self`, `new` c は無条件に一歩進む。合同的な場合（$+$, $<$, `if`, `let`, `set`）は、部分式に帰納法の仮定を当てて三択のまま持ち上げる。本質的なのは三箇所である。

■`get` —- 範囲外アクセスが起きない。 heap_ok の 1 より $|\Sigma| = |\Omega|$。仮定 $\Omega(o) = c$ より $o < |\Omega|$。よって $o < |\Sigma|$ であり、$\Sigma(o)$ が存在する。STGet が適用できる。

$$\dfrac{\dfrac{\Omega(o) = c \qquad |\Sigma| = |\Omega|}{\exists v.\ \Sigma(o) = v}\,\text{heap\_ok 1}}{H; o \vdash \texttt{get} \longrightarrow H; \epsilon; v}\,\textsc{STGet}$$

■$e_0 \Leftarrow m(e_1)$ —- メソッドが必ず見つかる。 e_0 に帰納法の仮定。値でなければ STSend1 / AwSend1。値なら型が `actor`$[c_1]$ だから標準形補題により $e_0 = o'$ かつ $\Omega(o') = c_1$。続いて e_1 に帰

納法の仮定。値なら STSend が適用できる —- その側条件 $\Omega(o') = c_1$ は標準形補題が**取り出した**もの、$\mathrm{mtab}(c_1, m) = (\tau_a, \tau_r)$ は HSend の側条件そのものである。

$$\dfrac{\dfrac{e_0 \text{ は値} \qquad \Omega;\Phi;c;\Gamma \vdash e_0 : \texttt{actor}[c_1]}{e_0 = o' \ \wedge\ \Omega(o') = c_1}\ \text{補題 18} \qquad \mathrm{mtab}(c_1, m) = (\tau_a, \tau_r)\ \ (\text{HSend の側条件}) \qquad e_1 \text{ は値}}{H; o \vdash o' \Leftarrow m(e_1) \ \longrightarrow\ (\Omega, \Sigma, \Phi \cdot \tau_r, \Psi \cdot \bot); \langle o' \Leftarrow m(e_1) \rhd |\Phi| \rangle; |\Phi|}\ \textsc{STSend}$$

■`await` e —- **ここで第三の枝が生まれる。** e に帰納法の仮定。値なら型が `future` τ だから標準形補題により $e = \kappa$ かつ $\Phi(\kappa) = \tau$。heap_ok の 2 より $|\Psi| = |\Phi|$ だから $\Psi(\kappa)$ は存在する。その中身で分かれる。

$$\dfrac{\Phi(\kappa) = \tau \qquad |\Psi| = |\Phi| \qquad \Psi(\kappa) = v}{H; o \vdash \texttt{await}\ \kappa \ \longrightarrow\ H; \epsilon; v}\ \textsc{STAwait}$$

$$\dfrac{\Phi(\kappa) = \tau \qquad |\Psi| = |\Phi| \qquad \Psi(\kappa) = \bot}{\mathrm{awaiting}(H, \texttt{await}\ \kappa)}\ \textsc{AwHere}$$

$\Psi(\kappa)$ **が「範囲外」になることはない**。$\bot$ か値かのどちらかである。これが第三の枝を「未解決 future の await」に限定できている理由である。 □

9.5 局所保存

補題 21 (局所保存)**.** heap_ok(H)、$\Omega(o) = c$、$\Omega;\Phi;c;\emptyset \vdash e : \tau$、$H; o \vdash e \ \longrightarrow\ H'; \mu; e'$ ならば

1. heap_ok(H')
2. $\Omega \sqsubseteq \Omega'$
3. $\Phi \sqsubseteq \Phi'$
4. $\Omega';\Phi';c;\emptyset \vdash e' : \tau$
5. $\forall M \in \mu.\ \mathrm{msg\_ok}(H', M)$

五つを**同時に**言うのが要点である。2, 3 が無いと、他のタスクや飛んでいるメッセージの型付けを H' に持ち上げられない。5 が無いと、送出されたメッセージが不変条件を満たすことが言えない。

Proof. $;\vdash \longrightarrow ;;$ の導出に関する帰納法。20 の規則すべてを扱う。代表的な四つを示す。

■STSet —- **状態の型不変条件を保つ。** 型付けの反転により $\tau = \texttt{unit}$ と $\Omega;\Phi;c;\emptyset \vdash v : \mathrm{stype}(c)$。補題 16 で $\forall C', \Gamma'$ 版に上げる。$H' = (\Omega, \Sigma[o \mapsto v], \Phi, \Psi)$。heap_ok$(H')$ の 3 を示すには、更新した位置とそれ以外で場合分けする。

$$\dfrac{\Sigma[o \mapsto v](o) = v \qquad v \text{ は値} \qquad \dfrac{\Omega(o) = c \qquad \Omega; \Phi; c; \emptyset \vdash v : \mathrm{stype}(c)}{\forall C', \Gamma'.\ \Omega; \Phi; C'; \Gamma' \vdash v : \mathrm{stype}(c)}\ \text{補題 16}}{v \text{ は値} \wedge \forall C', \Gamma'.\ \cdots \vdash v : \mathrm{stype}(c)}\ \text{heap\_ok 3（}o_2 = o\text{）}$$

$$\dfrac{\Sigma[o \mapsto v](o_2) = \Sigma(o_2) \qquad \text{heap\_ok}(H) \text{ の } 3}{\cdots}\ \text{heap\_ok 3（}o_2 \neq o\text{）}$$

Ω, Φ は変わらないので 2, 3 は反射律、4 は HUnit、5 は $\mu = \epsilon$ で空である。

■STNew **—- 表が伸びる。** $H' = (\Omega \cdot c_n,\ \Sigma \cdot \mathrm{sinit}(c_n),\ \Phi,\ \Psi)$。$\Omega \sqsubseteq \Omega \cdot c_n$ は追記の一般性質。$\text{heap\_ok}(H')$ の 1 は $|\Sigma \cdot x| = |\Sigma| + 1 = |\Omega| + 1 = |\Omega \cdot c_n|$。3 は、添字が元の範囲なら古い条件（ただし補題 15 で $\Omega \cdot c_n$ へ持ち上げ）、末尾なら SInitValue と SInitOk。4 も同様に持ち上げる。結論の式は $|\Omega|$ であり、$(\Omega \cdot c_n)(|\Omega|) = c_n$ だから HORef で $\texttt{actor}[c_n]$。

■STSend **—- 静的な型情報がメッセージに変わる瞬間。** 型付けの反転により、witness $c_1, \tau_{a1}, \tau_{r1}$ が出て、

$$\Omega; \Phi; c; \emptyset \vdash o' : \texttt{actor}[c_1], \qquad \mathrm{mtab}(c_1, m) = (\tau_{a1}, \tau_{r1}), \qquad \Omega; \Phi; c; \emptyset \vdash v : \tau_{a1}, \qquad \tau = \texttt{future}\ \tau_{r1}.$$

一方、遷移の側条件は $\Omega(o') = c$、$\mathrm{mtab}(c, m) = (\tau_a, \tau_r)$ である。HORef の反転から $\Omega(o') = c_1$ が出るので $c_1 = c$、したがって mtab の値も一致して $(\tau_{a1}, \tau_{r1}) = (\tau_a, \tau_r)$。$H' = (\Omega, \Sigma, \Phi \cdot \tau_r, \Psi \cdot \bot)$ で、新しい future の型は τ_r。

$$\dfrac{v \text{ は値} \qquad \dfrac{\Omega(o') = c \qquad \mathrm{mtab}(c, m) = (\tau_a, \tau_r) \qquad \Omega; \Phi; c; \emptyset \vdash v : \tau_a}{\forall C', \Gamma'.\ \Omega; \Phi \cdot \tau_r; C'; \Gamma' \vdash v : \tau_a}\ \text{補題 16 + 15} \qquad (\Phi \cdot \tau_r)(|\Phi|) = \tau_r}{\mathrm{msg\_ok}(H', \langle o' \Leftarrow m(v) \rhd |\Phi| \rangle)}\ \text{msg\_ok の構成}$$

$\text{heap\_ok}(H')$ の 4 では、新しく足した位置 $|\Phi|$ について $\Psi \cdot \bot$ の値が $\bot$ なので前提が成り立たず、自動的に満たされる。

■STAwait **—- future の型が返る。** 反転により $\Phi(\kappa) = \tau$。遷移の側条件は $\Psi(\kappa) = v$。$\text{heap\_ok}(H)$ の 4 がまさにこの二つを前提として「v は値かつ $\Omega; \Phi; ; \vdash v : \tau$」を与える。

$$\dfrac{\dfrac{\Phi(\kappa) = \tau \qquad \Psi(\kappa) = v}{v \text{ は値} \wedge \forall C', \Gamma'.\ \cdots \vdash v : \tau}\ \text{heap\_ok 4}}{\Omega; \Phi; c; \emptyset \vdash v : \tau}\ \text{STAwait の保存}$$

■合同規則（10 個）。 部分式に帰納法の仮定を当て、H' での型付けを得る。**もう一方の部分式**の型付けは H のものしか無いので、補題 15 で H' へ持ち上げてから規則を組み直す。たとえば STSend2 では：

$$\dfrac{\mathrm{mtab}(c_1, m) = (\tau_a, \tau_r) \qquad \dfrac{\dfrac{\Omega; \Phi; c; \emptyset \vdash v : \texttt{actor}[c_1]}{\Omega'; \Phi'; c; \emptyset \vdash v : \texttt{actor}[c_1]}\ \text{補題 15} \qquad \dfrac{\Omega; \Phi; c; \emptyset \vdash b : \tau_a \qquad H; o \vdash b \longrightarrow H'; \mu; b'}{\Omega'; \Phi'; c; \emptyset \vdash b' : \tau_a}\ \text{IH}}{\Omega'; \Phi'; c; \emptyset \vdash v \Leftarrow m(b') : \texttt{future}\ \tau_r}\ \textsc{HSend}$$

□

9.6 定理 8（保存）の証明

Proof. $\mathcal{C} \Longrightarrow \mathcal{C}'$ を反転して三つの場合に分ける。補助として、msg_ok と task_ok が表の拡張で保たれること（補題 15 から直ちに従う）を使う。

■CTask。 task_ok から、そのタスクの actor のクラス c と future の型 τ を取り出し、補題 21 を適用する。得られた五つがそのまま使える。古いメッセージ・古いタスクは 2, 3 で持ち上げ、新しいメッセージ μ は 5 から、更新されたタスクは 4 から。

■CFinish —- reply **が起こる場所。** $H' = (\Omega, \Sigma, \Phi, \Psi[\kappa \mapsto v])$。$\Omega$, Φ は不変なので型付けは一切変わらず、メッセージと他のタスクはそのままでよい。示すべきは heap_ok(H') の 4 だけである。

$$\dfrac{\dfrac{[\, o \vdash v \rhd \kappa \,]\ \text{が構成にある}}{\Phi(\kappa) = \tau\ \wedge\ \Omega; \Phi; c; \emptyset \vdash v : \tau}\ \text{task\_ok} \qquad v\ \text{は値} \qquad \Psi[\kappa \mapsto v](\kappa) = v}{v\ \text{は値} \wedge \forall C', \Gamma'.\ \cdots \vdash v : \tau}\ \text{heap\_ok 4（}\kappa_2 = \kappa\text{）}$$

task_ok が「タスクの式の型」と「解決すべき future の型」を最初から同一の τ として持っていたから、ここが埋まる。**これが reply の健全性である。**現行 AIPL のように両者が結ばれていなければ、この一点が埋まらない。

■CDeliver —- actor **起動。** ヒープは変わらない。示すべきは、新しく生まれるタスク $[\, o \vdash \mathrm{mbody}(c, m)[x_0 := v] \rhd \kappa \,]$ が task_ok を満たすこと。三つの事実が合流する。

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\text{msg\_ok} \Rightarrow \Omega(o) = c_0,\ \mathrm{mtab}(c_0, m) = (\tau_{a0}, \tau_{r0}),\ \Phi(\kappa) = \tau_{r0} \\ \textsc{CDeliver} : \Omega(o) = c,\ \mathrm{mtab}(c, m) = (\tau_a, \tau_r) \\ c_0 = c,\ (\tau_{a0}, \tau_{r0}) = (\tau_a, \tau_r)\end{array}}{\mathrm{mtab}(c, m) = (\tau_a, \tau_r)}\ \text{msg\_ok と側条件}}{\Omega; \Phi; c; x_0{:}\tau_a \vdash \mathrm{mbody}(c, m) : \tau_r}\ \textsc{BodiesOk} \qquad \forall C', \Gamma'.\ \cdots \vdash v : \tau_a\ (\text{msg\_ok より})}{\Omega; \Phi; c; \emptyset \vdash \mathrm{mbody}(c, m)[x_0 := v] : \tau_r}\ \text{補題 19}$$

そして $\Phi(\kappa) = \tau_r$ も msg_ok が持っているので、task_ok の三成分がそろう。msg_ok **が** *future* **の型まで運んでいたのはこのため**である。 □

9.7 定理 9（進行）の証明

補題 22 (タスク列の進行)**.** heap_ok(H) かつすべてのタスクが task_ok を満たすとき、次のいずれか：

1. タスク列を $\overline{t_1}, [\, o \vdash e \rhd \kappa \,], \overline{t_2}$ と分解できて、e が値であるか、e が一歩進む。
2. すべてのタスクの式が awaiting。

Proof. タスク列に関する帰納法。空列なら 2（前提が空虚）。先頭のタスクに補題 20 を適用して三択。値か一歩なら 1（$\overline{t_1}$ を空にとる）。awaiting なら残りに帰納法の仮定を当て、1 なら $\overline{t_1}$ の先頭に足し、2 ならそのまま 2。 □

定理 *9* の証明. 構成 $(H; \overline{M}; \overline{t}\,)$ について $\overline{M}$ で場合分け。

■メッセージがある場合 —– 必ず配送できる。 先頭を $\langle o \Leftarrow m(v) \rhd \kappa \rangle$ とすると、msg_ok から $\Omega(o) = c$ と $\mathrm{mtab}(c, m) = (\tau_a, \tau_r)$。これは CDeliver の側条件そのものである。

$$\dfrac{\dfrac{\mathrm{conf\_ok}(\mathcal{C}) \qquad \langle o \Leftarrow m(v) \rhd \kappa \rangle \in \overline{M}}{\Omega(o) = c \ \wedge\ \mathrm{mtab}(c, m) = (\tau_a, \tau_r)}\ \text{msg\_ok（定理 7）}}{\mathcal{C} \Longrightarrow (H;\ \overline{M'};\ \overline{t}, [\, o \vdash \mathrm{mbody}(c, m)[x_0 := v] \rhd \kappa \,])}\ \textsc{CDeliver}$$

型が無ければここが埋まらない。宛先のクラスが引けなければ、どのメソッド本体を起こせばよいか決まらない。定理 7 の内容がこの一点に効いている。

■メッセージが無い場合。 補題 22 を使う。

- 分解 1 が得られ、その式が値なら CFinish。
- 分解 1 が得られ、その式が一歩進むなら CTask。
- すべてが awaiting なら、タスク列が空なら終状態、空でなければブロック（第三の枝）。

□

9.8 定理 11–13 の証明

Proof. まず $\Longrightarrow^*$ の長さに関する帰納法で $\mathrm{conf\_ok}(\mathcal{C}) \wedge \mathcal{C} \Longrightarrow^* \mathcal{C}' \implies \mathrm{conf\_ok}(\mathcal{C}')$（定理 8 の反復）。

型安全性は、$\mathrm{conf\_ok}(\mathcal{C}')$ に定理 9 を適用すると終状態・一歩・ブロックのいずれかであり、stuck の定義がその三つすべてを否定しているので矛盾する。

状態の型不変条件と future の型不変条件は、$\mathrm{conf\_ok}(\mathcal{C}')$ の第一成分 $\mathrm{heap\_ok}(H')$ の 3, 4 そのものである。 □

10　補題の依存関係

定理・補題	依存
環境の外延性（補題 14）	なし
表の単調性（補題 15）	なし
値の文脈非依存性（補題 16）	なし
標準形（補題 18）	なし
代入（補題 19）	環境の外延性
局所進行性（補題 20）	標準形、heap_ok 1, 2
局所保存（補題 21）	代入、単調性、値の文脈非依存性、SInit*
タスク列の進行（補題 22）	局所進行性
定理 7（理解できない）	conf_ok の展開のみ
定理 8（保存）	局所保存、代入、BodiesOk、単調性
定理 9（進行）	タスク列の進行、標準形
定理 11（型安全性）	定理 8, 9
定理 12, 13（不変条件）	定理 8

11　保証しないこと

性質	成否	理由
デッドロック自由	一般には言えない	`await` がある。進行定理の第三の枝がそれ。ただし個別のプログラムについては証明できる（§12）
停止性	言えない	メソッドが自分あてに送れば無限に走る
公平性	言えない	どのメッセージを配送するかは非決定的
actor の逐次性	落とした	同じ actor あての二通が同時にタスクになりうる
合流性	言えない	証明していない
型推論アルゴリズムの健全性	言えない	型付け**関係**を定義しただけ。単一化は範囲外
実装との対応	言えない	OCaml 実装との refinement は証明していない

11.1　actor の逐次性について

ABCM/ABCL では各 actor が一度に一つのメッセージしか処理しない。AIPL$^-$ の CDeliver は無条件に新しいタスクを起こすので、同じ actor に対して複数のタスクが並行に走りうる。これは**型安全性には影響しない**—- heap_ok は状態が常に stype(c) 型であることしか言わず、どの順で書き換えられるかは問わないからである。しかし「`get` してから `set` する間に他のタスクが割り込まない」といった性質は言えない。逐次性を入れるには、構成に actor ごとの「実行中フラグ」を足し、CDeliver の側条件にする必要がある。

11.2　デッドロックについて

進行定理の第三の枝を消すには、送信の依存関係に順序を入れる必要がある。すなわち型に線形性やセッション型を入れる。先行レポートが構想していた静的セッション型

$$\Gamma; \Delta \vdash \texttt{send}\ a.m(\bar{e}) \triangleright \Delta'$$

はその方向である。ただしそれは本レポートの体系の**外**にある別の体系であり、「AIPL に型を付ければデッドロックが消える」わけではない。

12　哲学者の食事問題

型が付いてもデッドロックは排除されない（§11）。しかし**個別のプログラム**については証明できる。本節では哲学者の食事問題を AIPL^{-} で書き、デッドロックしないことを二つの層に分けて証明する。

12.1　プログラム

3 人の哲学者と 3 本のフォークを、それぞれ actor にする。フォークは「空きかどうか」だけを状態に持ち、哲学者は自分の段階を状態に持つ。

オブジェクト	クラス	状態の型	状態の意味
0, 1, 2	`Fork0..2`	`bool`	`true` = 空き
3, 4, 5	`Phil0..2`	`int`	0 = 思考中, 1 = lo 要求中, 2 = lo 保持・hi 要求中, 3 = 食事中

哲学者 i が使う 2 本を $\mathrm{lo}(i)$、$\mathrm{hi}(i)$ とし、**必ず番号の小さい方から取る**。

哲学者	使うフォーク	lo	hi	
0	0, 1	0	1	
1	1, 2	1	2	
2	2, 0	**0**	**2**	← ここが非対称。輪を切る

```
class Fork : bool { // true = 空き
  method req(p : int) : unit {
    if get() then { set(false); send phil(p).granted(()) }
            else { send phil(p).denied(()) }
  }
  method rel(u : unit) : unit { set(true) }
}

class Phil : int { // 0 思考 / 1 lo要求 / 2 lo保持 / 3 食事
  method go(u : unit) : unit {
    set(1); send fork(lo).req(me)
  }
  method granted(u : unit) : unit {
    if get() < 2 then { set(2); send fork(hi).req(me) } // lo が取れた
                 else { set(3); send self.eat(()) } // hi も取れた
```

```
  }
  method denied(u : unit) : unit { // 断られたら再要求
    if get() < 2 then send fork(lo).req(me)
                 else send fork(hi).req(me)
  }
  method eat(u : unit) : unit {
    set(0); send fork(lo).rel(()); send fork(hi).rel(()); send self.go(())
  }
}
```

send はすべて past 型であり、**await は一度も現れない**。これが第 1 層の証明の鍵になる。

初期構成 $\mathcal{C}_0$ は、3 人の哲学者に go を送った状態である。

$$\mathcal{C}_0 = (H_0;\ \langle 3 \Leftarrow \mathtt{go}(()) \rhd 0\rangle, \langle 4 \Leftarrow \mathtt{go}(()) \rhd 1\rangle, \langle 5 \Leftarrow \mathtt{go}(()) \rhd 2\rangle;\ \varepsilon)$$

補題 23 (プログラムが型検査を通る)**.** 上のクラス表は定義 *2* の三条件をすべて満たし、conf_ok($\mathcal{C}_0$) が成り立つ。

Coq では dp_sinit_value, dp_sinit_ok, dp_bodies_ok, dp_conf_ok として機械検証してある。18 個のメソッド本体すべてについて導出を構成した。

注意 24 (本体がオブジェクトを名前で参照すること)*.* 哲学者の本体には fork(lo) のようにフォークの**番号**が直接書き込まれる。これを許すため、BodiesOk（定義 2）を起動時のオブジェクト表 Ω_0 に相対化した。すなわち

$$\mathrm{mtab}(c,m) = (\tau_a,\tau_r) \implies \forall \Omega \sqsupseteq \Omega_0, \Phi.\ \Omega;\Phi;c;x_0{:}\tau_a \vdash \mathrm{mbody}(c,m) : \tau_r$$

とし、conf_ok に $\Omega_0 \sqsubseteq \Omega$ を加えた。Ω は伸びるだけなのでこの条件は保存される（定理 8 の証明に $\sqsubseteq$ の推移律を一箇所足すだけで済む）。

12.2 第 1 層 — 機械的デッドロック自由

進行定理（定理 9）の第三の枝、すなわち blocked に到達しないことを言う。

定義 25 (await を含まない式)**.** afree(e) を、e が await を部分式として含まないことと定義する。構成 $\mathcal{C}$ が afree であるとは、すべてのタスクの式が afree であることをいう。

補題 26.
1. 値は afree。
2. afree(v) かつ afree(e) ならば afree($e[x := v]$)。
3. awaiting(H, e) ならば $\neg$afree(e)。
4. heap_ok(H)、afree(e)、$H; o \vdash e \longrightarrow H'; \mu; e'$ ならば afree(e')。

4 が要点である。get が返すのはヒープ上の値であり、heap_ok の第 3 成分がそれを**値**だと保証するので 1 が使える。await の規則は前提が afree(e) と矛盾するので現れない。

補題 27 (afree の保存)**.** すべてのメソッド本体が afree ならば、conf_ok($\mathcal{C}$) かつ afree である構成の遷移先も afree である。

CDeliver で新しく生まれるタスク $\mathrm{mbody}(c,m)[x_0 := v]$ について、本体が afree、引数 v は msg_ok より値、したがって補題 26 の 1, 2 から afree。

定理 28 (await を含まないプログラムは詰まらない)**.** すべてのメソッド本体が afree で、conf_ok($\mathcal{C}$) かつ $\mathcal{C}$ が afree ならば、$\mathcal{C} \Longrightarrow^* \mathcal{C}'$ なるすべての $\mathcal{C}'$ について

$$\mathrm{conf\_ok}(\mathcal{C}') \ \wedge \ \neg\mathrm{blocked}(\mathcal{C}') \ \wedge \ (\mathrm{terminal}(\mathcal{C}') \vee \exists \mathcal{C}''.\ \mathcal{C}' \Longrightarrow \mathcal{C}'').$$

Proof. 保存（定理 8）と afree の保存を $\Longrightarrow^*$ の長さに関する帰納法で反復する。blocked ならタスクは空でなく、その先頭の式は awaiting かつ afree となり補題 26 の 3 に反する。最後に進行定理を適用すると、第三の枝が消えているので終状態か一歩進むかのどちらかになる。 □

系 29 (哲学者の食事問題はデッドロックしない)**.** $\mathcal{C}_0 \Longrightarrow^* \mathcal{C}'$ なるすべての $\mathcal{C}'$ について $\neg\mathrm{blocked}(\mathcal{C}')$ であり、$\mathcal{C}'$ は終状態であるか必ず一歩進める。

Coq では `dining_no_deadlock` である。

12.3 第 2 層 —- 資源デッドロック自由

第 1 層は「システムが止まらない」ことしか言わない。哲学者の食事問題で本当に問われるのは、**全員が 1 本ずつ持ったまま誰も食べられなくなる**という資源デッドロックである。これを、フォーク割り当てプロトコルの抽象状態機械について証明する。

■抽象状態。 n 人の哲学者と n 本のフォーク。哲学者 i は $\mathrm{lo}(i)$、$\mathrm{hi}(i)$ を使う。状態は

$$s = (\mathrm{phase} : \mathbb{N} \to \{\mathsf{Think}, \mathsf{Hold}, \mathsf{Eat}\}, \quad \mathrm{own} : \mathbb{N} \to \mathbb{N}_{\perp})$$

すなわち各哲学者の段階と、各フォークの保持者である。

■状態の健全性 $\mathrm{wf}(s)$**。**

- $\mathrm{own}(f) = i \implies i < n$
- $\mathrm{phase}(i) = \mathsf{Think} \implies i$ はどのフォークも持たない
- $\mathrm{phase}(i) = \mathsf{Hold} \implies \mathrm{own}(\mathrm{lo}(i)) = i$ かつ i が持つのは $\mathrm{lo}(i)$ のみ
- $\mathrm{phase}(i) = \mathsf{Eat} \implies \mathrm{own}(\mathrm{lo}(i)) = \mathrm{own}(\mathrm{hi}(i)) = i$

■遷移。

$$\frac{i < n \qquad \mathrm{phase}(i) = \mathsf{Think} \qquad \mathrm{own}(\mathrm{lo}(i)) = \perp}{s \to s[\mathrm{phase}(i) := \mathsf{Hold},\ \mathrm{own}(\mathrm{lo}(i)) := i]}\ \textsc{TakeLo}$$

$$\frac{i < n \qquad \mathrm{phase}(i) = \mathsf{Hold} \qquad \mathrm{own}(\mathrm{hi}(i)) = \perp}{s \to s[\mathrm{phase}(i) := \mathsf{Eat},\ \mathrm{own}(\mathrm{hi}(i)) := i]}\ \textsc{TakeHi}$$

$$\frac{i < n \qquad \mathrm{phase}(i) = \mathsf{Eat}}{s \to s[\mathrm{phase}(i) := \mathsf{Think},\ \mathrm{own}(\mathrm{lo}(i)) := \perp,\ \mathrm{own}(\mathrm{hi}(i)) := \perp]}\ \textsc{Release}$$

定義 30 (優先度規律)**.**

$$\mathrm{prio\_ordered}(n, \mathrm{lo}, \mathrm{hi}) \iff \forall i < n.\ \mathrm{lo}(i) < \mathrm{hi}(i)$$

定理 31 (資源デッドロック自由)**.** $\mathrm{prio\_ordered}(n, \mathrm{lo}, \mathrm{hi})$、$0 < n$、$\mathrm{wf}(s)$ ならば、s から進める遷移が必ず存在する。

Proof. Eat の哲学者がいれば RELEASE、Hold で hi が空いている者がいれば TAKEHI、Think で lo が空いている者がいれば TAKELO が使える。残るのは、Eat が一人もおらず、Hold の者は皆 hi を待たされ、Think の者は皆 lo を待たされている場合である。ここで矛盾を導く。

Hold の哲学者が一人もいなければ、Eat もいないので誰もフォークを持っていない。すると哲学者 0 は Think で $\mathrm{lo}(0)$ が空いており TAKELO が使える —- 仮定に反する。

Hold の哲学者がいるとき、その中で hi が**最大**の者を p とする。

$$\dfrac{\dfrac{\dfrac{p \text{ は Hold} \qquad \mathrm{own}(\mathrm{hi}(p)) \neq \bot}{\mathrm{own}(\mathrm{hi}(p)) = q}\ \text{仮定} \qquad \dfrac{q \text{ は Eat でない} \qquad \text{Think なら何も持たない}}{q \text{ は Hold}}\ \text{EAT も THINK もありえない}}{\dfrac{q \text{ が持つのは } \mathrm{lo}(q) \text{ のみ}}{\mathrm{hi}(p) = \mathrm{lo}(q)}\ \mathrm{wf} \qquad \dfrac{\mathrm{lo}(q) < \mathrm{hi}(q)}{\mathrm{hi}(p) < \mathrm{hi}(q)}\ \text{優先度規律} \qquad \dfrac{q \text{ は Hold}}{\mathrm{hi}(q) \leq \mathrm{hi}(p)}\ p \text{ の最大性}}\ \text{矛盾}}{\bot}$$

□

12.4 優先度規律は捨てられない

上の証明で使ったのは $\mathrm{lo}(i) < \mathrm{hi}(i)$ ただ一つである。これが本当に必要であることも証明できる。素朴な「左を取ってから右を取る」割り当て

$$\mathrm{lo}'(i) = i, \qquad \mathrm{hi}'(0) = 1,\ \mathrm{hi}'(1) = 2,\ \mathrm{hi}'(2) = 0$$

は $i = 2$ で $\mathrm{lo}'(2) = 2 \not< 0 = \mathrm{hi}'(2)$ となり規律を満たさない。そして

$$s_{\mathrm{dead}} = (\ \forall i.\ \mathrm{phase}(i) = \mathsf{Hold}, \quad \mathrm{own}(f) = f\ (f < 3)\ )$$

すなわち全員が自分の左のフォークを持った状態は wf を満たし、かつ**一歩も進めない**。

定理 32 (必要十分)**.** *1.* 本プログラムの lo, hi は優先度規律を満たし、よってどの wf 状態からも遷移できる。

2. 素朴な $\mathrm{lo}', \mathrm{hi}'$ は優先度規律を満たさず、s_{dead} は wf でありながら遷移を持たない。

Coq では `ordering_is_necessary_and_sufficient` である。

12.5 二つの層の間 —- 機械検証していない部分

注意 33 (正直に述べておくべきこと). 定理 28（系 29）は **AIPL$^-$ のプログラムそのもの**についての定理であり、意味論の中で機械検証されている。定理 31 は**フォーク割り当てプロトコル**についての定理で、これも機械検証されている。しかし

$$\text{プログラムの到達可能構成} \longrightarrow \text{抽象状態}$$

という射影が遷移を保つこと（精密化）は**機械検証していない**。対応は下表のとおりで、読めば明らかではあるが、証明はしていない。

抽象状態	プログラムの構成
$\mathrm{phase}(i) = \mathsf{Think}$	哲学者 i の状態が 0 または 1
$\mathrm{phase}(i) = \mathsf{Hold}$	哲学者 i の状態が 2
$\mathrm{phase}(i) = \mathsf{Eat}$	哲学者 i の状態が 3
$\mathrm{own}(f) = i$	フォーク f の状態が `false` で、それを取ったのが i
TakeLo	`req` が `granted` を返す（1 回目）
TakeHi	`req` が `granted` を返す（2 回目）
Release	`eat` が 2 本を `rel` する

なお、フォークの状態は `bool` しか持たないので「誰が持っているか」はプログラム上には現れない。精密化を機械検証するなら、まずフォークの状態を保持者つきに変える（あるいは補助的な履歴変数を入れる）必要がある。

13　セッション型は使えるか

13.1　二者間セッション型で言えること

フォークとのやりとりは、二者間セッション型としてきれいに書ける。

$$\texttt{Fork} \;=\; \&\big\{\, \texttt{req}(\texttt{int}).\ \oplus\{\, \texttt{granted}.\mathbf{end},\ \texttt{denied}.\mathbf{end}\,\},\quad \texttt{rel}(\texttt{unit}).\mathbf{end}\,\big\}$$

哲学者側はその双対 $\overline{\texttt{Fork}}$ をたどる。これで保証されるのは**プロトコル遵守**である —- `granted` を受け取っていないのに `rel` を送る、`req` に `unit` を渡す、といった誤りが静的に排除される。

AIPL の現行実装が持つ `protocol_define` / `protocol_check_send` は、まさにこれを**実行時**に検査している。先行レポートの Coq 形式化（`coq/AbclSoundness.v`）は、その線形オートマトンとしての性質（順序保存・線形消費・完了後拒否・違反拒否）を証明している。したがって「セッション型を静的にする」という道筋自体は、既に半分できている。

13.2　しかしデッドロックは排除できない

注意 34 (哲学者の食事問題はセッション型の標準的な反例). 二者間セッション型は**一つのセッションの中**での順序しか見ない。哲学者 i とフォーク $\mathrm{lo}(i)$ のセッション、哲学者 i とフォーク $\mathrm{hi}(i)$ のセッション —- どちらも完全に型が付く。デッドロックは**セッションどうしの絡み**から生じるので、二者間セッション型では見えない。定理 32 の s_{dead} がまさにそれで、すべてのセッションは型どおりに進行しているのに全体は止まる。

デッドロックまで型で排除するには、次のいずれかが要る。

■(a) 線形論理にもとづくセッション型。 Caires–Pfenning、Wadler の CP などは、プロセス網が**木状（非巡回）**であることを型付けの側で強制し、その代償としてデッドロック自由を構成的に得る。しかし哲学者の食事問題は**環**であり、この体系では**そもそも型が付かない**。安全性を証明するのではなく、プログラムを拒否する。したがって我々の目的には合わない。

■(b) **マルチパーティセッション型。** Honda–Yoshida–Carbone の体系なら、環全体を一つの大域型として書き、各参加者へ射影できる。単一のマルチパーティセッションの中ではデッドロック自由が言える。ただし全員のやりとりを一つのセッションにまとめる必要があり、AIPL の「actor が独立にメッセージを送り合う」モデルとは相性が悪い。中央の振り付け（choreography）を書くことになる。

■(c) **優先度つきセッション型。** Kobayashi の usage 型 (obligation/capability レベル)、Padovani、Dardha–Gay の priority-based session types は、各チャネル・各資源に**優先度**を与え、**優先度の昇順にしか獲得できない**ことを型規律にする。環状のプロセス網も型が付き、しかもデッドロック自由が保証される。

13.3 本レポートの証明は (c) そのものである

注意 35. §12 の prio_ordered、すなわち $\mathrm{lo}(i) < \mathrm{hi}(i)$ は、**フォーク f の優先度を f とした優先度規律**にほかならない。そして定理 31 の証明（「待たれている優先度の最大値を取ると矛盾する」）は、この種の型体系の健全性証明の標準的な議論そのものである。定理 32 は、その規律が必要でもあることを示している。

つまり**「セッション型は使えるか」への答えは、使える、ただし優先度つきのものである**。そして本レポートは、その型体系を AIPL$^-$ に実装する代わりに、**意味論の側で同じことを証明した**ことになる。

AIPL$^-$ に優先度を型として載せるなら、次の形になる。actor 型に優先度を付け、獲得済み優先度の上界 π を型判断に持たせる。

$$\frac{\Omega;\Phi;C;\Gamma \vdash e : \texttt{actor}[c]^{\,\rho} \qquad \pi < \rho \qquad \mathrm{mtab}(c,m) = (\tau_a,\tau_r) \qquad \Omega;\Phi;C;\Gamma \vdash e_1 : \tau_a}{\Gamma \vdash_{\pi} e \Leftarrow m(e_1) : \texttt{future}\ \tau_r \ \rhd\ \rho}\ \textsc{HAcquire}$$

すなわち「いま持っている最大の優先度 π より真に大きい優先度の資源しか取れない」。この規則で型が付いたプログラムに対して定理 31 の議論がそのまま通り、**デッドロック自由が型から従う**。これを AIPL$^-$ 全体に対して形式化するのが自然な次の一歩である。

13.4 現行実装への含み

1. いまの `protocol_define` は actor.method の**線形列**である。これを二者間セッション型（選択 $\oplus$ と分岐 &、再帰を持つもの）に一般化すれば、`granted`/`denied` の分岐が書ける。
2. さらに actor（あるいはロック・資源）に**優先度**を宣言できるようにし、送信規則で昇順を強制すれば、デッドロック自由が静的に得られる。文法上は `class Fork : bool @ priority 0 { ... }` のような注釈で足りる。
3. この二段階は独立に導入できる。1 だけでもプロトコル誤りは静的に消える。

14 実装へのフィードバック

1. **メソッド宣言に返り値型を入れる。**これが最優先である。`src/ast.ml` の `method_decl` に返り値型を足し、`src/infer.ml` の `preinfer_all_classes` が登録するメソッド型を $(\overline{\tau}) \to \tau_r$ にする。現在は $\to$ `unit` 固定である。
2. **`reply` を本体の値と結ぶ。**最小の変更は、`reply(v)` を型検査時に「v の型がメソッドの返り値型と単一化する」という制約にすることである。AIPL$^-$ のように文法から消してしまう必要はない。
3. **そうすれば `now` / `future` の `any` が消える。**`now t.m(e) :` τ_r、`future t.m(e) : future` τ_r、`await e :` τ が閉じる。定理 13 がその保証を与える。
4. **クラスの状態にも型を宣言する。**現在の `var count = 0;` は初期値から型を推論しているが、宣言があれば heap_ok の 3 に直接対応する。
5. `send!`、remote、`sender` は `any` 境界として残してよい。ただし**境界であることを型で明示**し、静的断片との区別を保つべきである。

参考文献

[1] Akinori Yonezawa, Jean-Pierre Briot, Etsuya Shibayama. Object-Oriented Concurrent Programming in ABCL/1. *OOPSLA 1986*, pp. 258–268.

[2] Kenjiro Taura, Satoshi Matsuoka, Akinori Yonezawa. ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language. 1994.

[3] Naoki Kobayashi, Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. *OOPSLA 1994*.

[4] Andrew K. Wright, Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation* 115(1), 1994.

[5] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.（store typing と参照の扱いは第 13 章に従う）

[6] Luís Caires, Frank Pfenning. Session Types as Intuitionistic Linear Propositions. *CONCUR 2010*.（線形論理にもとづくセッション型。木状の網に限りデッドロック自由）

[7] Philip Wadler. Propositions as Sessions. *ICFP 2012*.

[8] Kohei Honda, Nobuko Yoshida, Marco Carbone. Multiparty Asynchronous Session Types. *POPL 2008*.

[9] Naoki Kobayashi. A Type System for Lock-Free Processes. *Information and Computation* 177(2), 2002.（優先度（レベル）による型システム。本レポート §13 の (c)）

[10] Ornela Dardha, Simon J. Gay. A New Linear Logic for Deadlock-Free Session-Typed Processes. *FoSSaCS 2018*.

[11] `docs/ABCLCP_TYPE_SOUNDNESS_REPORT.md`（2026-05-04）。本レポートはその第 9 節が挙げた拡張 1, 3, 4, 5 を実行したものである。

付録 A　Coq 形式化の対応表

本文	coq/AIPLSoundness.v
型 τ	`Inductive ty := TInt` TBool | TUnit | TActor | TFut|
式 e	`Inductive tm`
値	`Inductive value`
$e[x := v]$	`Fixpoint subst`
ヒープ $(\Omega, \Sigma, \Phi, \Psi)$	`Record heap := Heap { hot; hst; hft; hfv }`
メッセージ・タスク・構成	`Definition msg / task / conf`
$\Omega \sqsubseteq \Omega'$	`Definition ext`
stype, sinit, mtab, mbody	`Variable stype / sinit / mtab / mbody`
定義 2	`Hypothesis sinit_value / sinit_ok / bodies_ok`
$\Omega; \Phi; C; \Gamma \vdash e : \tau$	`Inductive ht`
$H; o \vdash e \longrightarrow H'; \mu; e'$	`Inductive tstep`
awaiting	`Inductive awaiting`
$\mathcal{C} \Longrightarrow \mathcal{C}'$	`Inductive cstep`
$\Longrightarrow^*$	`Inductive csteps`
heap_ok, msg_ok, task_ok, conf_ok	`Definition heap_ok / msg_ok / task_ok / conf_ok`
stuck, blocked, terminal	`Definition stuck / blocked / terminal`
補題 14	`ht_env_ext`
補題 15	`ht_mono`
補題 16	`value_ht_indep`
補題 18	`canon_int / canon_bool / canon_actor / canon_fut`
補題 19	`substitution`
補題 20	`local_progress`
補題 21	`local_preservation`
補題 22	`tasks_progress`
定理 7	`no_method_not_understood`
定理 8	`preservation`, `preservation_star`
定理 9	`progress`
定理 11	`type_safety`
定理 12	`state_type_invariant`
定理 13	`future_type_invariant`
哲学者の食事問題　coq/AIPLDining.v	
プログラム（クラス表）	`dp_stype / dp_sinit / dp_mtab / dp_mbody`
補題 23	`dp_sinit_value / dp_sinit_ok / dp_bodies_ok / dp_conf_ok`
初期構成 $\mathcal{C}_0$	`dp_C0`
afree	`Inductive afree`, `conf_afree`

本文	coq/AIPLSoundness.v
補題 26	value_afree / afree_subst / afree_not_awaiting / afree_tstep
定理 28	async_deadlock_free
系 29	dining_no_deadlock
抽象状態・遷移	Record pst, Inductive pstep, wf
優先度規律	prio_ordered
定理 31	no_dead_state, priority_implies_deadlock_free
定理 32	ordering_is_necessary_and_sufficient
詰まり状態 s_{dead}	deadS, deadS_wf, deadS_stuck

検証環境: Coq / Rocq Prover 9.1.0。AIPLSoundness.v が 1121 行、AIPLDining.v が 566 行。補題・定理は合わせて 70 個。上記すべてについて Print Assumptions が *Closed under the global context* を返す。