AIPL の型健全性と型安全性（第 2 版）

— 現行実装から証明可能な範囲 AIPL^{-2} を切り出し、効果と期限つきで証明する —

児玉工房 / AICE · AIPL プロジェクト

2026 年 07 月 30 日 14:02

概要

第 1 版（2026 年 7 月 28 日）は、`reply` と戻り値型が結ばれていない当時の実装から、それを結んだ**仮想の**部分言語 AIPL^{-} を切り出して型健全性と型安全性を述べた。その後の実装でこの仮定は現実になり、さらに効果注釈と期限付き待ちが加わった。

本稿は二段構えである。第 I 部で**現行実装から証明が成立する範囲を切り出す**。どの機能が入り、どれが外れ、なぜそうなるかを実装の事実に基づいて判定する。第 II 部でその範囲 **AIPL^{-2}** を効果と期限つきで定式化し、第 III 部で証明する。

主結果は**進行定理が二択になる**ことである。第 1 版の進行定理は「終状態・簡約できる・未解決 future の await でブロック」の三択であり、第三の枝がデッドロックであった。AIPL^{-2} では期限のない待ちが構文に存在せず、待ちは残り時間を減らす簡約として進むので、**ブロックの枝が消える**。系として**デッドロック自由**が言える。これは第 1 版が「一般には言えない」と明記した性質である。代償は明確で、停止性は依然として言えない。

副産物として、効果の健全性から**実時間側のアクターがエージェント側を待てない**ことが定理として出る。また定式化の過程で、現行実装の効果伝播に穴があることが分かった —- `now` を `future` と `await` に分けて書くと効果検査を逃れる。第 14.1 節で報告する。

目次

第 I 部　証明が成立する範囲を切り出す 2

1 **第 1 版との関係** 2

2 **現行実装の機能一覧と可否判定** 3

3 **入る理由・外れる理由** 4

3.1 期限のない待ちを外すと何が変わるか 4

3.2 プリミティブをどう扱うか 4

3.3 指定エラー —- 配列の範囲外 5

3.4 アクター型が区別されない件 5

4 **AIPL^{-2} の確定** 5

第 II 部　定式化 5

5 構文 5
5.1 糖衣 6
6 表と構成 6
7 型付け規則 6
8 操作的意味論 7
9 不変条件 8
第 III 部　証明 8
10 補題 8
11 主定理 9
12 効果の健全性と、その系 10
13 言えないこと 11
第 IV 部　実装との対応 11
14 モデルと実装のずれ 11
14.1 実装の効果伝播に穴がある 11
14.2 差分テストが埋めている部分 12
15 Coq 形式化の見通し 12
16 まとめ 13

第 I 部
証明が成立する範囲を切り出す

1　第 1 版との関係

第 1 版 `docs/aipl_soundness_ja.pdf`（28 ページ）は次の立場を採った。

> 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、その型健全性と型安全性を Coq で機械検証した。

その後の実装で状況が変わった。第 1 版が「一点だけ直した仮想の言語」として仮定したものが、実装済みになっている。

第 1 版が仮定または課題としたこと	現在	実装
`reply` を本体の値と結ぶ	**実装済**	ρ 表による推論と注釈
`now` / `future` の `any` を消す	**実装済**	送信地点が ρ を返す
クラスの状態に型を宣言する	一部	初期値から推論。宣言はまだ
`send!`・remote・`sender` を any 境界として残す	そのまま	変わらず
効果	**追加**	8 種の効果注釈と検査
期限付き待ち	**追加**	`now/await ... timeout k else e`
返信の線形性	**追加**	高々一度、注釈時はちょうど一度

したがって本稿の AIPL^{-2} は、第 1 版の AIPL^{-} と違って**仮想の言語ではない**。現行実装の部分集合であり、その部分集合に留まって書けば定理が保証を与える。

注意 1. 第 1 版は公開済み（`https://kodamay.org/reports/2026-07-28_aipl_soundness.pdf`）なので上書きせず、本稿を第 2 版として並置する。第 1 版の定理は AIPL^{-} について依然として正しい。本稿は対象を広げ、主張を一つ強める。

2 現行実装の機能一覧と可否判定

現行実装（`yaskodama/abclcp`, branch `make-src-base`）の機能を一つずつ判定する。判定の根拠は実装の事実であり、推測ではない。

機能	判定	理由
変数・局所束縛 `var x = e`	**入る**	代入補題の対象
算術・比較・等値・単項マイナス	**入る**	`/` は `int` 同士でも `float`、整数除算は `div`。評価器と型が一致している
真偽値リテラル・条件分岐	**入る**	基本
文字列と `++`	**入る**	`++` は全域関数（両辺を文字列化）。失敗しない
フィールド（状態）	**入る**	store の型不変条件を言うため
クラス表・メソッド表	**入る**	メソッド解決の核
`new C` による生成	**入る**	表が動的に伸びる。単調性が要る
`future` 送信	**入る**	中心
`await ... timeout k else e`	**入る**	**本稿の中心**
`now ... timeout k else e`	**入る**	上の糖衣
`send`（past 型）	**入る**	future を捨てる糖衣
`reply`	**入る**	戻り値型と結ばれた。線形性も検査される
戻り値型注釈	**入る**	$\mathrm{ret}(C, m)$ として表に入れる
効果注釈	**入る**	**本稿で追加**
`self`	**入る**	状態アクセスとクラス文脈
プリミティブ（`print`, `ai_call` 等）	**入る**	宣言された型と効果を持つ不透明な定数として扱う
期限のない `now` / `await`	**外れる**	**これを外すことが本稿の要点**。実装は既定で警告、`AIOS_STRICT_DEADLINE=1` でエラー
`sender`	外れる	型が `any` で、送信先のメソッドも検査されない
`remote(...)`	外れる	戻り値が `any`。対向ノードのインタフェース記述の機構が無い
`send!`（unsafe send）	外れる	明示的な escape hatch
`become`	外れる	置換後のインタフェース互換性を検査していない

機能	判定	理由
アクター型の区別	外れる	`unify` が `TActor(_,_), TActor(_,_) -> ()` で全アクターを同一視する
`select`	部分的に外れる	`case` の照合と `reply` の帰属は実装済みだが、受信の非決定性を構成に入れると議論が別問題になる
配列	条件つき	添字が範囲外なら実行時に失敗する。§3.3 の**指定エラー**として扱う
`while`	入るが停止性は言えない	ループ本体の `reply` は線形性検査が弾く
overload の曖昧性	外れる	注釈で一意に決まる範囲に限る。実装は曖昧をエラーにしている
多相（型スキーム）	外れる	引数は単相。単一化アルゴリズムの健全性は別課題
`express` / `atomic` / `where`	該当なし	未実装

3　入る理由・外れる理由

判定のうち、理由を述べておくべきものを挙げる。

3.1　期限のない待ちを外すと何が変わるか

第 1 版の進行定理の第三の枝は「すべてのタスクが未解決 future を await している」であった。期限がなければ、その状態から抜ける手段が言語に無い。だから型では排除できない。

期限を必須にすると事情が変わる。待ちは**残り時間を減らす**という簡約になり、残り時間が尽きれば `else` 節へ簡約される。つまり**待っているタスクは常に簡約できる**。第三の枝が空になり、進行定理は二択になる。

注意 2. これは「時間が経つ」ことを簡約として数える立場である。計算は進んでいないが構成は動いている。実装との対応は良い —- `await_future_timeout` は OCaml の `Condition` に時限待ちが無いため 1 ミリ秒刻みのポーリングで実装されており、**文字どおり**時間を刻む簡約になっている。

3.2　プリミティブをどう扱うか

`ai_call` や `print` の中身は AIPL の外にある。これらを**宣言された型と効果を持つ不透明な定数**として扱う。すなわち $p : (\bar{\tau}) \to \tau \,!\, \varepsilon_p$ が与えられており、$p(\bar{v})$ は 1 ステップで τ 型の値へ簡約され、その際 ε_p の効果を起こす、と公理化する。

これは逃げではなく、効果系の意味そのものである。効果の健全性（定理 17）は「プログラムが宣言した以外の効果を起こさない」を主張するが、プリミティブが宣言どおりに振る舞うことは前提であり、証明できない。**前提であることを明示するのが本稿の立場**である。

3.3　指定エラー —- 配列の範囲外

配列の添字が範囲外なら実行時に失敗する。これを詰まり（stuck）と呼んでしまうと型安全性が壊れるので、標準的な扱いに従い**指定エラー err** を導入する。型安全性の主張は「型の不整合で詰まらない」であり、**err** への到達は認める。ゼロ除算と同じ扱いである。

3.4　アクター型が区別されない件

実装の `unify` は `TActor` 同士を無条件に成功させる。したがって `actor[A]` と `actor[B]` が単一化できてしまう。これは潜在的な穴であり、AIPL^{-2} では**アクター型はクラス名で区別される**という前提を置く。実装がこの前提を満たしていないので、この一点は**モデルが実装より厳しい**箇所である（§14）。

4　AIPL^{-2} の確定

以上より、AIPL^{-2} は次の言語である。

> 現行 AIPL のうち、`sender`・`remote`・`send!`・`become`・`select` を使わず、すべての `now` と `await` に期限と `else` 節を書き、すべてのメソッドに戻り値型と効果を注釈し、アクター型をクラス名で区別する言語。

この範囲に留まって書けば、以下の定理が保証を与える。

第 II 部
定式化

5　構文

$$
\begin{aligned}
&\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{string} \mid \texttt{actor}[C] \mid \texttt{future}\langle \tau \,!\, \varepsilon \rangle \\
&\varepsilon \subseteq \mathcal{E} = \{\texttt{mut}, \texttt{time}, \texttt{io}, \texttt{mem}, \texttt{net}, \texttt{ai}, \texttt{fs}, \texttt{log}\} \\
&v ::= n \mid b \mid () \mid s \mid a \mid f \\
&e ::= v \mid x \mid e \oplus e \mid \texttt{var}\ x = e;\ e \mid \texttt{if}\ e\ e\ e \\
&\qquad \mid \texttt{self.fld} \mid \texttt{self.fld} := e \mid \texttt{new}\ C(\bar{e}) \mid p(\bar{e}) \\
&\qquad \mid \texttt{future}\ e.m(\bar{e}) \mid \texttt{await}\ e\ \texttt{timeout}\ k\ \texttt{else}\ e \mid \texttt{reply}\ e
\end{aligned}
$$

a はアクター名、f は future 名、$k \in \mathbb{N}$ は残り時間である。$\texttt{future}\langle \tau \,!\, \varepsilon \rangle$ は「τ 型の値を返す future で、それを待つと ε の効果を負う」を表す。

5.1 糖衣

$$\texttt{now}\ e.m(\bar{e})\ \texttt{timeout}\ k\ \texttt{else}\ e_a \ \equiv\ \texttt{var}\ f = \texttt{future}\ e.m(\bar{e});\ \texttt{await}\ f\ \texttt{timeout}\ k\ \texttt{else}\ e_a$$
$$\texttt{send}\ e.m(\bar{e}) \ \equiv\ \texttt{var}\ \_ = \texttt{future}\ e.m(\bar{e});\ ()$$

注意 3 (効果の伝播がこの糖衣から導かれる). 効果を future の型に載せたので、**待つ側だけが呼ばれる側の効果を負う**という規律が規則を追加せずに出る。`now` は `await` を含むので負い、`send` は future を捨てるので負わない。第 1 版に無かった論点であり、実装が `now` の辺に沿って不動点で伝播させているのを、より一様な形で述べたことになる。

6 表と構成

$\mathrm{mtab}(C, m) = (\bar{\tau}) \to \tau_r \,!\, \varepsilon_m$	メソッド表
$\mathrm{ret}(C, m) = \tau_r$、$\mathrm{eff}(C, m) = \varepsilon_m$	戻り値型と宣言効果
$\mathrm{stype}(C)$	クラス C の状態の型
$\Omega : a \mapsto C$	アクター表
$\Sigma : a \mapsto v$	状態表
$\Phi : f \mapsto (\tau, \varepsilon)$	future 型表
$\Psi : f \mapsto v_\perp$	future 値表（$\perp$ は未解決）
M	メッセージの多重集合 $\langle a \Leftarrow m(\bar{v}),\ f \rangle$
T	タスク列 (a, e, f)。f は解決すべき future

構成は $\mathcal{C} = \langle \Omega, \Sigma, \Phi, \Psi, M, T \rangle$ である。

7 型付け規則

判断は $\Omega; \Phi; \Gamma; C \vdash e : \tau \,!\, \varepsilon$ である（C は `self` のクラス、Γ は変数環境）。主要な規則のみ示す。

$$\frac{\mathrm{stype}(C) = \tau}{\Omega;\Phi;\Gamma;C \vdash \texttt{self.fld} : \tau \,!\, \emptyset}\ \text{T-Field} \qquad \frac{\mathrm{stype}(C) = \tau \qquad \Gamma \vdash e : \tau \,!\, \varepsilon}{\Gamma;C \vdash \texttt{self.fld} := e : \texttt{unit} \,!\, \varepsilon \cup \{\texttt{mut}\}}\ \text{T-Assign}$$

$$\frac{\mathrm{mtab}(C', \texttt{init}) = (\bar{\tau}) \to \texttt{unit} \,!\, \_ \qquad \Gamma \vdash e_i : \tau_i \,!\, \varepsilon_i}{\Gamma \vdash \texttt{new}\ C'(\bar{e}) : \texttt{actor}[C'] \,!\, \bigcup \varepsilon_i \cup \{\texttt{mem}\}}\ \text{T-New}$$

$$\frac{p : (\bar{\tau}) \to \tau \,!\, \varepsilon_p \qquad \Gamma \vdash e_i : \tau_i \,!\, \varepsilon_i}{\Gamma \vdash p(\bar{e}) : \tau \,!\, \bigcup \varepsilon_i \cup \varepsilon_p}\ \text{T-Prim}$$

$$\frac{\Gamma \vdash e : \texttt{actor}[C'] \,!\, \varepsilon_0 \qquad \mathrm{mtab}(C', m) = (\bar{\tau}) \to \tau_r \,!\, \varepsilon' \qquad \Gamma \vdash e_i : \tau_i \,!\, \varepsilon_i}{\Gamma \vdash \texttt{future}\ e.m(\bar{e}) : \texttt{future}\langle \tau_r \,!\, \varepsilon' \rangle \,!\, \varepsilon_0 \cup \bigcup \varepsilon_i}\ \text{T-Future}$$

$$\frac{\Gamma \vdash e : \texttt{future}\langle \tau \,!\, \varepsilon' \rangle \,!\, \varepsilon_0 \qquad \Gamma \vdash e_a : \tau \,!\, \varepsilon_a \qquad k > 0}{\Gamma \vdash \texttt{await}\ e\ \texttt{timeout}\ k\ \texttt{else}\ e_a : \tau \,!\, \varepsilon_0 \cup \varepsilon' \cup \varepsilon_a}\ \text{T-Await}$$

$$\frac{\mathrm{ret}(C, m) = \tau \qquad \Gamma \vdash e : \tau \,!\, \varepsilon}{\Gamma; C; m \vdash \texttt{reply}\ e : \texttt{unit} \,!\, \varepsilon}\ \text{T-Reply}$$

$$\frac{\Gamma, \bar{x}{:}\bar{\tau}; C; m \vdash e_{\mathrm{body}} : \texttt{unit} \,!\, \varepsilon \qquad \varepsilon \subseteq \mathrm{eff}(C, m) \qquad \mathrm{replies}(e_{\mathrm{body}}) = 1}{\vdash \texttt{method}\ m(\bar{x}) : \tau_r \,!\, \mathrm{eff}(C, m)\ \{e_{\mathrm{body}}\}}\ \text{T-Method}$$

T-Await が本稿の中心である。三点に注意されたい。

1. $k > 0$ を要求する。期限 0 の `await` は**型付け規則の側では**書けない。$k = 0$ は簡約の途中でのみ現れる中間状態である。
2. `else` 節は同じ τ を持たなければならない。期限切れでも式全体の型は変わらない。
3. 効果 ε' —- 呼ばれる側の宣言効果 —- を負うのは **await する側**である（注意 3）。

8 操作的意味論

タスク内の簡約 $\Sigma; e \to \Sigma'; e'; acts$ は通常の評価文脈で与える。構成の簡約 $\mathcal{C} \Longrightarrow \mathcal{C}'$ のうち、本稿で重要なものを示す。

$$\frac{\langle a \Leftarrow m(\bar{v}), f \rangle \in M \qquad \Omega(a) = C \qquad \mathrm{mtab}(C, m) \text{ の本体を } e_0 \text{ とする}}{\langle \Omega, \Sigma, \Phi, \Psi, M, T \rangle \Longrightarrow \langle \Omega, \Sigma, \Phi, \Psi, M \setminus \{\cdot\}, (a, e_0[\bar{x} := \bar{v}], f) :: T \rangle}\ \text{C-Deliver}$$

$$\frac{\Psi(f) = \bot}{\langle \ldots, (a, \texttt{reply}\ v, f) :: T \rangle \Longrightarrow \langle \ldots, \Psi[f \mapsto v], \ldots, T \rangle}\ \text{C-Reply}$$

$$\frac{\Psi(f') = v}{\langle \ldots, (a, \texttt{await}\ f'\ \texttt{timeout}\ k\ \texttt{else}\ e_a, f) :: T \rangle \Longrightarrow \langle \ldots, (a, v, f) :: T \rangle}\ \text{C-Resolve}$$

$$\frac{\Psi(f') = \bot \qquad k > 0}{\langle \ldots, (a, \texttt{await}\ f'\ \texttt{timeout}\ k\ \texttt{else}\ e_a, f) :: T \rangle \Longrightarrow \langle \ldots, (a, \texttt{await}\ f'\ \texttt{timeout}\ (k-1)\ \texttt{else}\ e_a, f) :: T \rangle}\ \text{C-Tick}$$

$$\frac{\Psi(f') = \bot}{\langle \ldots, (a, \texttt{await}\ f'\ \texttt{timeout}\ 0\ \texttt{else}\ e_a, f) :: T \rangle \Longrightarrow \langle \ldots, (a, e_a, f) :: T \rangle}\ \text{C-Timeout}$$

定義 4 (待ちは常に進む)**.** C-Resolve、C-Tick、C-Timeout の三つは $\Psi(f')$ と k の値によって**網羅的かつ排他的**である。解決済みなら C-Resolve、未解決で $k > 0$ なら C-Tick、未解決で $k = 0$ なら C-Timeout。したがって `await` を先頭に持つタスクは**必ず**簡約できる。

これが第 1 版との唯一の本質的な違いであり、進行定理が二択になる理由のすべてである。

9 不変条件

定義 5 (構成の健全性)**.** conf_ok($\mathcal{C}$) を次の合成とする。

- heap_ok：$\forall a.\ \Omega(a) = C \implies \Sigma(a)$ は stype(C) 型。
- fut_ok：$\forall f.\ \Phi(f) = (\tau, \varepsilon)$ かつ $\Psi(f) = v \neq \bot \implies v$ は τ 型。
- msg_ok：$\langle a \Leftarrow m(\bar{v}), f\rangle \in M$ ならば $\Omega(a) = C$ に m があり、$\bar{v}$ は mtab(C, m) の引数型を持ち、$\Phi(f) = (\mathrm{ret}(C, m), \mathrm{eff}(C, m))$。
- task_ok：$(a, e, f) \in T$ ならば $\Omega(a) = C$ として $\Omega; \Phi; \emptyset; C \vdash e : \texttt{unit}\,!\,\varepsilon$ かつ $\varepsilon \subseteq \mathrm{eff}(C, m_f)$。ここで m_f は f を生んだメソッド。

task_ok の後半が第 1 版に無い条件である —- **走っているタスクが今後起こしうる効果は、そのメソッドが宣言した効果の範囲に収まっている**。

第 III 部
証明

10 補題

第 1 版の補題（型環境の外延性、表の単調性、値の型付けの文脈独立性、標準形、代入）はそのまま使える。効果を足したことで一つ増える。

補題 6 (効果の単調性)**.** $\Gamma \vdash e : \tau\,!\,\varepsilon$ かつ $\Sigma; e \to \Sigma'; e'; acts$ ならば、ある $\varepsilon' \subseteq \varepsilon$ が存在して $\Gamma \vdash e' : \tau\,!\,\varepsilon'$ であり、かつ $acts$ が実際に起こす効果は ε に含まれる。

Proof. e の構造に関する帰納法。要点のみ。

- T-Prim：$p(\bar{v}) \to v'$ のとき、結果 v' は値なので $\varepsilon' = \emptyset$。実際に起こる効果は $\varepsilon_p \subseteq \varepsilon$。
- T-Assign：`self.fld` $:= v \to ()$。$\varepsilon' = \emptyset$、起こる効果は $\{\texttt{mut}\} \subseteq \varepsilon$。
- T-Await：C-Resolve なら $e' = v$ で $\varepsilon' = \emptyset$。C-Timeout なら $e' = e_a$ で $\varepsilon' = \varepsilon_a \subseteq \varepsilon$。C-Tick なら e' の型付けは k を除いて同一なので $\varepsilon' = \varepsilon$。いずれも $\subseteq$。
- 条件分岐は選ばれた枝の効果が全体の部分集合。
- 評価文脈は帰納法の仮定から。

効果は合併でのみ構成されるので、部分項が値へ潰れるたびに単調に減る。 □

注意 7. C-Tick で $\varepsilon' = \varepsilon$ となり**減らない**点は重要である。$\subseteq$ を主張しているので問題ないが、効果の減少を停止性の尺度に使うことはできない。停止性が言えない理由の一つがここにある。

11　主定理

定理 8 (理解できないメッセージは飛ばない)**.** conf_ok($\mathcal{C}$) かつ $\mathcal{C} \Longrightarrow^* \mathcal{C}'$ ならば、$\mathcal{C}'$ のメッセージはすべて msg_ok を満たす。すなわち存在しないメソッドへの送信も、引数の型が合わない送信も現れない。

定理 9 (保存 —- 型と効果)**.** conf_ok($\mathcal{C}$) かつ $\mathcal{C} \Longrightarrow \mathcal{C}'$ ならば conf_ok($\mathcal{C}'$)。とくに task_ok の効果条件が保たれる —- タスクの残余効果は宣言効果の部分集合のままである。

Proof. 簡約規則ごとの場合分け。第 1 版の局所保存補題に、補題 6 による効果条件の維持を加える。C-Deliver では msg_ok から $\Phi(f) = (\mathrm{ret}(C,m), \mathrm{eff}(C,m))$ が分かるので、T-Method の前提 $\varepsilon \subseteq \mathrm{eff}(C,m)$ が新しいタスクの効果条件をちょうど与える。C-Reply では fut_ok が T-Reply の前提から従う。表が伸びる規則（C-Deliver と `new`）については単調性補題を使う。 □

定理 10 (進行 —- 二択)**.** conf_ok($\mathcal{C}$) ならば、次のいずれかが成り立つ。

1. $\mathcal{C}$ は終状態である（M も T も空)。
2. ある $\mathcal{C}'$ が存在して $\mathcal{C} \Longrightarrow \mathcal{C}'$。

Proof. $M \neq \emptyset$ なら C-Deliver が適用できる。$M = \emptyset$ かつ $T \neq \emptyset$ とし、先頭タスク (a, e, f) を見る。task_ok より e は `unit` 型で型付く。標準形補題より e は値か、簡約基（redex）を評価文脈に持つ。

値の場合、$e = ()$ でタスクは終了できる（タスク除去規則)。簡約基を持つ場合、その形で場合分けする。第 1 版と異なるのは `await` の場合だけである。定義 4 により、C-Resolve、C-Tick、C-Timeout のいずれかが必ず適用できる。他の簡約基（算術、条件分岐、フィールド、`new`、プリミティブ、`future` 送信、`reply`）については第 1 版の局所進行性補題がそのまま使える。配列の範囲外は **err** への簡約であり、簡約できることに変わりはない。

よって $T \neq \emptyset$ ならつねに簡約できる。$M = \emptyset$ かつ $T = \emptyset$ が終状態である。 □

系 11 (デッドロック自由)**.** conf_ok($\mathcal{C}$) かつ $\mathcal{C} \Longrightarrow^* \mathcal{C}'$ ならば、$\mathcal{C}'$ は終状態か、簡約できる。すなわち ***AIPL*$^{-2}$ のプログラムはデッドロックしない**。

Proof. 定理 9 と 10 を $\Longrightarrow^*$ に沿って繰り返す。 □

注意 12 (第 1 版との差、そして限界). 第 1 版は「デッドロック自由は一般には言えない —- `await` がある」と明記し、進行定理の第三の枝としてそれを認めた。AIPL^{-2} では期限のない `await` が構文に無いので枝が消える。

ただし**これは「待たなくなる」という意味ではない**。相互に `now` で待ち合う二つのアクターは、両方が期限まで C-Tick を刻み、両方が `else` 節へ落ちる。プログラムは止まらないが、意図した結果も得られない。**デッドロックが有限時間の失敗に変換される**のであって、失敗が消えるのではない。これが期限を必須にすることの本質である。

定理 13 (型安全性)**.** stuck($\mathcal{C}$) を「終状態でなく、**err** でもなく、簡約もできない」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 14 (状態の型不変条件)**.** 到達可能なすべての構成で heap_ok が成り立つ。すなわちアクターの状態は常に $\mathrm{stype}(C)$ 型である。

定理 15 (future の型不変条件 — `reply` と戻り値型の一致)**.** 到達可能なすべての構成で fut_ok が成り立つ。すなわち $\Psi(f) = v$ ならば v は $\Phi(f)$ の第一成分の型を持つ。

注意 16. 定理 15 が、実装で見つけた不整合に対応する定理である。整数演算が `VFloat` を返していたとき、`reply(1+2)` は `int` と型付けされながら実行時に浮動小数を `resolve` していた。これは fut_ok の破れであり、定理 15 が成り立たない状態だった。**定理が実装のバグを名指しできる**という例である。

12 効果の健全性と、その系

定理 17 (効果の健全性)**.** $\mathrm{conf\_ok}(\mathcal{C})$ かつ $\mathcal{C} \Longrightarrow^{*} \mathcal{C}'$ とする。このとき、実行のあいだにアクター a（クラス C、メソッド m を処理中）が起こす効果はすべて $\mathrm{eff}(C, m)$ に含まれる。

Proof. task_ok の効果条件（$\varepsilon \subseteq \mathrm{eff}(C, m_f)$）が定理 9 で保たれる。補題 6 より、各ステップで実際に起こる効果はそのステップ直前の残余効果 ε に含まれる。$\varepsilon \subseteq \mathrm{eff}(C, m_f)$ なので従う。プリミティブが宣言どおりの効果しか起こさないことは前提である（§3.2）。 □

系 18 (機外に出ないことが言える)**.** $\texttt{net} \notin \mathrm{eff}(C, m)$ ならば、C の m を処理している間、そのアクターはノード外へ通信しない。

系 19 (実時間側はエージェント側を待てない)**.** $\mathrm{eff}(C, m) \cap \{\texttt{net}, \texttt{ai}, \texttt{fs}\} = \emptyset$（実時間プロファイルの条件）を満たすメソッド m の本体では、$\mathrm{eff}(C', m') \ni \texttt{net}$ なるメソッドを ***await***（したがって ***now***）できない。

Proof. T-Await により、待つ側の効果は $\varepsilon' = \mathrm{eff}(C', m')$ を含む。T-Method の前提 $\varepsilon \subseteq \mathrm{eff}(C, m)$ に反する。一方 `send` は future を捨てるので ε' を負わず、許される。 □

注意 20. 系 19 は、先の言語仕様レポートで「境界の方向の規律」として設計判断として述べたものである。効果を future の型に載せると、**規則を追加せずに定理として出る**。「制御ループがモデルの返りを待って期限を落とす」という最も起こりやすい事故が、T-Await 一つから排除される。

13　言えないこと

性質	成否	理由
デッドロック自由	**言える**	定理 11。第 1 版から変わった点
効果の上界	**言える**	定理 17
境界の方向	**言える**	系 19
停止性	言えない	`while` があり、自分あて送信で無限に走れる。C-Tick も効果を減らさない
応答時間の上界	言えない	期限は**待つ側**の上界であり、計算の上界ではない
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。第 1 版と同じ
意図した結果が得られること	言えない	期限切れは `else` 節へ落ちる。詰まらないが正しくもない
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
プリミティブの効果宣言の正しさ	前提	証明できない（§3.2）
範囲外の機能	対象外	`sender`、remote、`become`、`select`、`send!`

第 IV 部
実装との対応

14　モデルと実装のずれ

	モデル	実装
期限のない `await`	存在しない	既定で警告。`AIOS_STRICT_DEADLINE=1` でエラー
アクター型の区別	クラス名で区別	`unify` が全アクターを同一視（**実装が緩い**）
効果の伝播	future の型に載る	`now` の辺に沿って不動点（**実装が緩い**、§14.1）
待ちの刻み	C-Tick	1 ミリ秒のポーリング（対応が良い）
逐次性	落としている	1 アクター 1 スレッドで逐次（**実装が厳しい**）
配列の範囲外	**err**	実行時例外（対応する）

14.1　実装の効果伝播に穴がある

所見 21. 定式化の過程で、現行実装の効果伝播に穴があることが分かった。実装は `now` 送信の地点で「呼ぶ側 → 呼ばれる側」の辺を張り、その辺に沿って不動点まで効果を伝播させている。しかし `await` では伝播させていない。したがって

```
var f = future planner.plan(g);            // 辺が張られない
var r = await f timeout 100 else "";       // ここでも伝播しない
```

と書くと、`now` を使った場合と**意味は同じ**なのに`{ai, net}` が伝播せず、効果検査を逃れる。

再現を確認した。同一のクラスに二つのメソッドを置き、どちらも `!{time, log}` と宣言する。

```
class Planner {
```

```
  method plan(g) : string !{ai, net} { reply(ai_call("plan " ++ g)); }
}
class Ctrl {
  var p = new Planner();
  method viaNow() : unit !{time, log} {              // 弾かれる（正しい）
    var r = now p.plan("x") timeout 100 else "";
    print(r);
  }
  method viaFutureAwait() : unit !{time, log} {      // 通ってしまう
    var f = future p.plan("x");
    var r = await f timeout 100 else "";
    print(r);
  }
}
```

viaNow だけを残して型検査すると

```
TYPE_ERROR: line 7, col 10: method Ctrl.viaNow declares {log, time}
  but its body has {ai, log, net} (missing {ai, net})
```

と正しく弾かれる。ところが viaFutureAwait だけを残すと

```
OK
```

となる。系 19（実時間側はエージェント側を待てない）が**実装では now と書いたときにしか効いていない**。本稿の定式化はこの穴を持たない。効果を future$\langle\tau\,!\,\varepsilon\rangle$ として future の型に載せているので、await の規則が必ず ε を負わせる。now が糖衣であることから伝播が導かれるので、future+await に分けても逃れられない。

実装の修正方針は明確である。TFuture of ty を TFuture of ty * eff に変え、FutureSend が呼ばれる側の宣言効果を型に載せ、Await がそれを加える。現在の now_edges と不動点は不要になる（future の型を通じて自然に伝播するため）。

14.2 差分テストが埋めている部分

第 1 版は「実装との対応は言えない —- OCaml 実装との refinement は証明していない」と述べた。これは今も正しい。ただし部分的な検査は常設した —- 型検査が付けた戻り値型と、実行時に reply された値の型を突き合わせるハーネスである（scripts/type_runtime_diff.py）。

これは定理 15（future の型不変条件）を**到達した構成についてだけ**確かめる仕組みである。証明の代わりにはならないが、実際にこれで整数演算の fut_ok 破れを検出した。

15 Coq 形式化の見通し

第 1 版の Coq 開発（変数・局所束縛・代入補題、状態 store と型不変条件、クラス表とメソッド表、メールボックスと非同期送信、future 表と await、動的生成）は、本稿の骨格をすでに含んでいる。追加が必要なのは次の三点である。

1. **効果**を型と対にして持つ。判断を $\vdash e : \tau\,!\,\varepsilon$ にし、task_ok に効果条件を足す。効果集合は有限なので list か有限集合で足りる。補題 6 が新規の証明義務である。

2. **期限**を `await` に持たせ、C-Tick / C-Timeout を加える。進行定理の `await` の場合が**易しくなる**（三つの規則が網羅的なので、ブロックの場合分けが消える）。
3. `future`$\langle \tau \,!\, \varepsilon \rangle$ への変更。future 型表 Φ が対を持つようになる。

注意 22. 作業量の見積もりとして、第 1 版の進行定理は `await` でブロックする場合の扱いが最も込み入っていた。本稿ではそこが単純化されるので、**効果を足す手間と、進行の証明が軽くなる分が概ね釣り合う**と見ている。

16 まとめ

1. 第 1 版は**仮想の**部分言語について証明した。本稿の AIPL^{-2} は**現行実装の部分集合**である。この範囲に留まって書けば定理が保証を与える。
2. 範囲の切り方（第 I 部）は実装の事実に基づく。入るもの・外れるものを一覧にし、外れる理由を述べた。外れるのは `sender`、remote、`send!`、`become`、`select`、そして**期限のない待ち**である。
3. 主結果は**進行定理が二択になる**こと。期限のない待ちを構文から外すと、待ちが簡約として進むのでブロックの枝が消える。系として**デッドロック自由**が言える—- 第 1 版が「一般には言えない」と明記した性質である。
4. ただしこれは**デッドロックが有限時間の失敗に変換される**ということであり、失敗が消えるのではない。相互に待ち合う二者は両方が期限切れになる。詰まらないが正しくもない。
5. 効果を future の型に載せると、**待つ側だけが呼ばれる側の効果を負う**という規律が規則を追加せずに出る。その系として**実時間側がエージェント側を待てない**ことが定理になる。
6. 定式化が実装の穴を見つけた。`now` を `future`+`await` に分けると効果検査を逃れる。修正方針は `TFuture` に効果を持たせることである。

参考

- 第 1 版 `docs/aipl_soundness_ja.pdf` — AIPL^{-} の定式化と Coq 検証、哲学者の食事問題、「保証しないこと」の一覧。`https://kodamay.org/reports/2026-07-28_aipl_soundness.pdf`
- `docs/aipl_reply_inference_ja.pdf` — `reply` からの戻り値型推論と注釈、線形性、整数演算の fut_ok 破れの発見と修正。
- `docs/aipl_abcl1_features_ja.pdf` — `where` と express mode、二つのプロファイル、理想仕様案。
- `docs/aipl_kobayashi_synthesis_ja.pdf` — 線形型・デッドロック自由な型・汎用型システム・資源使用型の位置づけ。
- Andrew K. Wright, Matthias Felleisen. *A Syntactic Approach to Type Soundness.* Information and Computation 115(1), 1994.
- Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.（store typing は第 13 章）