AIPL / ABCL^c+ ユーザーズガイド

— 文法・使い方・サンプルプログラム —

ABCL^c+ / AIOS 処理系（`yaskodama/abclcp`, branch `make-src-base`）

2026 年 08 月 23 日版

対象: branch `make-src-base`（七つの規律を含む現行仕様）

概要

本書は AIPL（ABCL^c+）の**現状の**ユーザーズガイドである。第 I 部で言語の全体像と最短の動かし方、第 II 部で文法と型システムの仕様、第 III 部でその上に載る七つの規律を、第 V 部でサンプルプログラムをソースと実行結果つきで、第 VI 部でビルド・実行・検査の道具立てを、第 VII 部に文法の要約・現状の限界・旧版からの移行を置く。

AIPL は**アクター**を単位とする並行言語で、値は `return` ではなく `reply` で返す。その上に三つの静的な仕組みが載っている — **型**（`reply` から戻り値型を推論し、注釈も書ける）、**効果**（`!{ai, net}` のようにメソッドが世界に及ぼす作用を宣言する）、**期限**（`now ... timeout ... else ...` で待ちに上限を置く）。たとえば「実時間で動くアクターが、機外のモデルを同期で呼ぶ」ことは、効果検査だけで禁止できる。

さらにその上に**七つの規律**が載っている（第 III 部）— 失敗の型 `result`$\langle\tau\rangle$、返信先の一級性、資源の順序、義務レベル、セッション型である。どれも**注釈を書かなくても効く**ように作ってある。その結果、**待ちの宛先がクラスとして分かるか、宛先ノードのレベルが宣言されている範囲では、デッドロック自由が型で保証される**。これは紙の上だけの主張ではなく、Coq/Rocq で機械検証してある。

本書が述べるのは、三つの処理系（OCaml 版・Python 版・JavaScript 版）すべてが備える**核**である。Python 版はこれに記録型・精緻化型・チャネル・所有などの**拡張**を持っており、言語は二層になっている（§43）。

前版（2026 年 07 月 30 日版）からの差分だけを知りたい場合は §43 を見ればよい。`become` は**言語から外した**。

目次

第 I 部
はじめに

1　AIPL とは

AIPL は ABCL/1 の系譜にあるアクター言語である。プログラムは**クラス宣言**と**グローバル文**の並びで、クラスから生成されたアクターがメッセージを送り合って動く。

```
class Counter {
  var n = 0;                        // フィールド（アクターの状態）
  method bump() : int !{mut} {      // 戻り値型 int、効果は mut
    n = n + 1;
    reply(n);                       // 呼び出し側へ値を返す
  }
}

var c = new Counter();              // アクターを1台つくる
```

```
print(now c.bump() timeout 100 else 0);
```

他の言語との違いのうち、最初に押さえるべきものは四つある。

1. **アクターは 1 台 1 スレッド**で、受け取ったメッセージを逐次に処理する。フィールドへの同時アクセスは起こらない。
2. **値は `reply` で返す。**`return` は無い。`reply` は呼び出し側の future を解決する操作で、メソッドの実行はその後も続けられる。
3. **送信には三種類ある。**`send`（待たない・文）、`now`（返信まで待つ・式）、`future`（待たずに引換券を得る・式）。
4. **待つと固まる。**`now` で待っている間、そのアクターは他のメッセージを一切処理できない。だから待ちには期限を書く。

2　三つの静的な仕組み

2.1　型 — 何を返すか

メソッドの戻り値型は本体の `reply` から推論される。省略可能な注釈 `method m(x) : T` も書ける。注釈を書くと、全実行パスで `reply` することが検査され、かつ宣言した型が**期待型**として式の推論に流れる。

2.2　効果 — 世界に何をするか

メソッドが持つ作用を `{...}` で宣言する。効果は 8 種類ある（§11）。省略すると本体から推論され、書けば「本体の効果 $\subseteq$ 宣言した効果」が検査される。

```
method plan(goal) : string !{ai, net} { reply(ai_call("plan: " ++ goal)); }
method peek()     : int    !{}        { reply(n); }
```

`ai` と `net` を分けてあるのが要点で、「AI を使うが機外へは出ない」オンデバイス推論を `{ai}` だけで表せる。

2.3　期限 — いつまで待つか

```
var r = now planner.plan(g) timeout 2000 else "(timed out)";
```

期限内に返信が来なければ `else` 節の値になる。`else` を書かなければ `result`$\langle\tau\rangle$ が返り、確かめずには使えない（§12）。

2.4　そのうえに七つの規律がある

型・効果・期限は土台である。その上に、失敗の型・返信先の一級性・資源の順序・義務レベル・セッション型が載っている（§III）。どれも**注釈を書かなくても効く**ように作ってある。その結果、**待ちの宛先がクラスとして分かるか、宛先ノードのレベルが宣言されている範囲では、デッドロック自由が型で保証される**（§15）。

3　5 分で動かす

3.1　処理系のビルド

```
cd ~/aios/abclcp
make -C src thread_repl
```

dune でのビルドは src/lexer.ml の二重生成で失敗するので、src/Makefile を使う。

3.2　プログラムを走らせる

REPL コマンドを書いたファイルを -f に渡す。

```
cat > run.repl <<'EOF'
load docs/samples/guide/g2_now_future.aipl
compile
EOF
./src/abclrepl_thread -q -f run.repl < /dev/null
```

```
twice = 43
awaited = 20
v=7
```

-f に .aipl を直接渡してはいけない。REPL コマンドとして 1 行ずつ解釈されてクラス定義が読まれず、Actor xxx not found になる。必ず load と compile を書く。

3.3　型検査だけを走らせる

REPL は SDL に依存するので、型検査専用の小さなドライバを別に用意している（§36）。ビルドすると ./tc file.aipl で戻り値型・効果・シグネチャの表が出る。

```
$ ./tc docs/samples/guide/g6_effects.aipl
=== g6_effects.aipl ===
[OK] type check passed
[method return types (inferred from reply)]
  Counter#bump -> int
  Planner#plan -> string
  ViaNow#run -> string
[method effects (inferred / declared)]
  Counter#bump !{mut}
  Planner#plan !{ai, net}
  ViaNow#run !{ai, net}
```

第 II 部
言語ガイド

4　プログラムの構成

プログラムは**宣言の並び**である。宣言は次のいずれかで、順序は自由だが、参照する名前は実行時に解決される。

宣言	内容
`class C { ... }`	クラス定義
`var x = e;`	グローバル変数
`var x = new C(args);`	アクターの生成
`send t.m(args);`	グローバルな非同期送信
`send! t.m(args);`	同上（検査を緩めた送信）
`f(args);`	組込み関数の呼び出し

グローバルの位置に `if` や `while` は書けない。制御構造が必要ならメソッドの中に置いて `send` で起動する。

クラスは**フィールドの並び**に**メソッドの並び**が続く形である。

```
class Counter {
  var n = 0;                    // フィールドは先。var か float で宣言する
  float ratio = 1.5;
  method init(start) {          // init は new のときに自動で呼ばれる
    n = start;
  }
  method bump() : int !{mut} {
    n = n + 1;
    reply(n);
  }
}
```

- フィールドはメソッドより**前**に書く。
- クラスには**メソッドが最低 1 つ**必要である（文法上、フィールドだけのクラスは書けない）。
- `init` という名前のメソッドは `new C(args)` のときに自動で呼ばれる。引数の個数が合わないと生成が飛ばされ、`[Actor] x: no init; skipped` と表示される。
- フィールドの初期化式で型が決まる。`var n = 0;` は `int` になるので、あとから `float` や文字列を代入すると型エラーになる。

5 字句

種別	内容
コメント	`// 行末まで`、`/* ... */`（入れ子にはできない）
予約語	`class`, `method`, `var`, `float`, `new`, `send`, `send!`, `now`, `future`, `await`, `if`, `else`, `while`, `do`, `select`, `case`, `timeout`, `call`, `replyto`, `self`, `sender`, `remote`, `true`, `false`
整数	`0`, `123`
浮動小数	`1.5`, `100.`（末尾のドットだけでも `float`）
文字列	`"..."`。`\\ \" \n \t \r` が使える
真偽値	`true`, `false`
識別子	`[A-Za-z_][A-Za-z0-9_]*`

`float` は予約語なので、変数名や型名として自由には使えない（戻り値型注釈では `: float` と書ける。文法規則を別に持っている）。

6 型

型	説明
`int`, `float`, `string`, `bool`, `unit`	基底型
`T[]`	配列（`array_*` 組込みで扱う）
`future T`	`future` 送信の結果。`await` で `T` を取り出す
`actor(C)`	クラス `C` のアクター。`new C(...)` の型
`any`	静的に内容を保証しない境界（`sender`、リモート送信先）

戻り値型注釈に書けるのは `int` / `float` / `string` / `bool` / `unit` / `any` と、大文字で始まるクラス名（アクター型）である。小文字で始まる未知の名前は `unknown type name in return annotation` として構文エラーになる。

メソッドの引数は単相である。シグネチャの引数型変数は全呼び出し地点で共有されるので、同じメソッドを `int` と `string` で呼ぶと衝突する。多相なメソッドは書けない。

`int` と `float` の間に**暗黙変換は無い**。`var h = new Hello(5);` が `float` のフィールドに繋がると型エラーになるので、`new Hello(5.)` と書く。

7 式

式	意味
`123`, `1.5`, `"abc"`, `true`, `false`	リテラル
`x`	変数・フィールド・仮引数
`-e`	単項マイナス。`neg(e)` へ落ちる。`int`/`float`
`a ++ b`	文字列連結（両辺を文字列化）
`a + b`, `a - b`, `a * b`	数値演算。`int` 同士は `int`、`float` 同士は `float`
`a / b`	除算。`int` 同士でも `float`（整数除算はしない）
`a == b`, `a != b`	等値・非等値。`int`/`float`/`string`/`bool`
`a > b`, `a < b`, `a >= b`, `a <= b`	比較。結果は `bool`
`new C(args)`	アクターの生成（効果 `{mem}`）
`now t.m(args)`	同期送信。返信が来るまで待ち、その値になる
`now t.m(args) timeout` n `else` e	期限付きの同期送信
`future t.m(args)`	非同期送信。`future T` を返す
`await e`	future の解決を待つ
`await e timeout` n `else` e'	期限付きの `await`
`f(args)`	組込み関数の呼び出し
`( e )`	括弧

送信先 `t` は**アクター参照が入った変数名**か `remote("host:port", "actorName")` である。クラス名ではない。

7.1 優先順位

弱い順に

等値 `== !=` ＜ 比較 `> < >= <=` ＜ 連結 `++` ＜ 加減 `+ -` ＜ 乗除 `* /` ＜ 単項マイナス

である。したがって `1 + 2 == 3` は `(1 + 2) == 3`、`"x=" ++ a + b` は `"x=" ++ (a + b)` と読まれる。二項演算子は**すべて左結合**である（等値も含む）— `1 == 2 == false` は `((1 == 2) == false)` と読まれて `true` になる。単項マイナスは前置なので重ねられる（`- - 3` は `3`）。

7.2 `+` と `++` を混同しない

`+` は**数値専用**である。文字列を混ぜると `no overload of + matches (string, int)` になる。

```
print("count = " ++ n);      // 正しい
print("count = " + n);       // 型エラー
```

`++` は両辺を文字列化してから連結するので、どちらが文字列でなくてもよい。

7.3 曖昧なオーバーロード

両辺が未束縛の `a + b` は `int` とも `float` とも解釈でき、principal type を持たない。これはエラーである。注釈を書けば期待型が式へ流れて一意に決まる。

```
method add(a, b)       { reply(a + b); }   // エラー: ambiguous overload
method add(a, b) : int { reply(a + b); }   // 注釈で確定する
```

`AIOS_LAX_OVERLOAD=1` で従来の「警告して既定候補を選ぶ」動作に戻せる。等値 `== !=` は候補が 4 つあるが戻り値がすべて `bool` なので曖昧にはならない（曖昧判定は**戻り値型が割れたとき**だけ行う）。

8 文

文	意味
`var x = e;`	変数宣言
`var x = new C(args);`	アクター生成つき宣言
`x = e;`	代入
`f(args);` `call f(args);`	組込み関数の呼び出し
`send t.m(args);`	非同期送信。返信を待たない
`send self.m(args);`	自分自身への非同期送信
`send sender.m(args);`	送信元への非同期送信
`send! t.m(args);`	検査を緩めた送信
`if (e) s` `if (e) s else s`	条件分岐（括弧が要る）
`while e do s`	繰り返し（括弧は要らない）
`{ s ... }`	ブロック
`select { ... }`	メッセージの選択受信

`now self.m()` と `now sender.m()` は**文法上書けない**。自分への `now` は自己デッドロックになる（自分は今そのメソッドを処理中なので、返信を作る主体がいない）ので、書けないことがそのまま安全側に働いている。幾何計算などを別に切り出したいときは、別アクターに置いて `now kin.solve(...)` と呼ぶ。

`var x : T = e;` という型注釈つきの変数宣言は**まだ無い**。仕様案にはあるが未実装である。

9 アクターと通信

9.1 三つの送信

	待つか	式か文か	呼ばれる側の効果を負うか
`send a.m(x);`	待たない	文	負わない
`now a.m(x)`	返信まで待つ	式（結果は宣言／推論された型）	**負う**
`future a.m(x)`	待たない	式（`future T`）	負わない

「待つ側だけが呼ばれる側の効果を負う」というのが効果伝播の規律である（§11）。

9.2 reply

メソッドは値を「返す」のではなく `reply(v)` で呼び出し側の future を解決する。型システムはメソッドごとに戻り値型を持ち、本体のすべての `reply` と、すべての `now` / `future` 送信地点がこの型を共有する。

- `reply` は**高々一度**。2 度目は待ち手がすでに 1 度目の値を受け取ったあとに来るので黙って捨てられる。これは型エラーにしてある。
- 戻り値型を `unit` 以外で**宣言**した場合は、**全実行パスでちょうど一度**になることが検査される。`while` の本体は 0 回実行されうるので、保守的に「reply しないパスがある」と判定される。
- `reply` がまったく無いメソッドは戻り値 `unit` と扱われる。`send` で呼ぶ「通知」型のメソッドがこれにあたる。

```
method m(k) : int { reply(k); reply(k + 1); }                    // 型エラー
method m(k) : int { if (k > 0) { reply(1); } else { reply(2); } } // OK
```

9.3 select

```
select {
  case job(k) -> { reply("done:" ++ k); }
  timeout 500 -> { print("timed out"); }
}
```

`case m(x1, ..., xn)` は同名メソッドの署名に照合される。引数の個数と型が合わないと型エラーになる。`timeout` の単位はミリ秒で、省略できる。

case 本体の `reply` は、囲むメソッドではなく「選択されたメッセージ」への返信である。処理系が case の実行前に返信先を選択されたメッセージへ切り替えるためで、上の例の `reply` は `job` への返信になる。したがって `select` を含むメソッド自身は `unit` でよい。一方 `timeout` 本体は囲むメソッドの返信先のまま実行される。

> `select` は**まだ処理されていない**メッセージを mailbox から探す。通常のメソッド dispatch が先に同名メッセージを取ってしまうと case には残らないので、`select` で待つ名前は直接呼ばせない設計にするのが安全である（§26 の実行結果を参照）。

10 期限付きの待ち

10.1 なぜ必要か

`now` や `await` を期限なしで書くと、返信が来なければ永久に待つ。アクターは 1 台 1 スレッドで、待ちは条件変数でブロックするので、待っている間そのアクターは他のメッセージを一切処理できない。石になる。

```
now t.m(a) timeout <ミリ秒> else <式>
await f    timeout <ミリ秒> else <式>
```

期限内に返信が来ればその値、来なければ `else` 節を評価した値になる。

10.2 規則

- `else` 節は**本体と同じ型**でなければならない。期限切れでも式全体の型は変わらないためである。違うと `now: the else branch has type string but the call returns int`。
- 期限は**正**でなければならない（`timeout must be positive`）。
- 期限なしの `now` / `await` は既定では**警告**で、`AIOS_STRICT_DEADLINE=1` で**エラー**になる。
- `reply` 回数の上界は、本体と `else` の**どちらか一方**が走るとみなして `max` を取る（和ではない）。
- 拒否された future（`reject`）は期限切れとは区別され、例外になる。

実装では OCaml の `Condition` に時限待ちが無いので、1 ms 刻みのポーリングで期限を測っている。

10.3 型健全性との関係

期限を書くと、待ちは有限時間で終わる。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**ということであって、失敗が消えるわけではない。互いに `now` で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。

詰まらないこと自体を型で言うには、期限ではなく**義務レベル**が要る（§15）。そちらは **Coq/Rocq で機械検証してある** — `await` を含んだままの中核について、型健全性・型安全性・デッドロック自由の三つを、公理なしで証明した（`coq/AIPLSoundness2.v`）。第 1 版は `await` を言語から取り除いた断片でしかデッドロック自由を言えていなかったので、そこが第 2 版の要点である。

進行定理は次のように変わった。

	進行定理の主張
第 1 版	終状態 ∨ 一歩進める ∨ **全タスクが待ち**
第 2 版	終状態 ∨ 一歩進める

停止性は依然として言えない。`while 1 > 0 do {}` は止まらないが、それは待ちではないので、進行定理とは矛盾しない。

11 効果

11.1 8 種類の効果

効果	意味	効果	意味
`mut`	フィールドへの代入	`net`	ノード外への通信
`time`	時刻の取得・待機	`ai`	モデル推論
`io`	装置への入出力	`fs`	永続化
`mem`	動的割り付け	`log`	観測のみの出力

これは新しく設計した分類ではなく、評価器が実行時メタデータとして持っていた capability 分類をそのまま静的にしたものである。`ai` と `net` を分けたのが要点で、機外へ出るモデル呼び出しは両方を持つが、オンデバイス推論には `{ai}` だけを与えればよい。「AI を使うが機外へは出ない」がシグネチャ

に現れる。

11.2　書き方

```
method plan(goal) : string !{ai, net} { ... }   // 戻り値型のあと
method peek()             !{}          { ... }   // 効果なしの明示
method tick()                          { ... }   // 省略すると推論
```

注釈は省略できる（gradual）。省略すれば推論だけが行われ、何も言われない。書けば「本体の効果 ⊆ 宣言した効果」が検査される。多めに宣言するのは可。未知の効果名は誤記として弾かれる。

```
[Type error] line 1, col 18: unknown effect(s) network in A.m;
  known effects are mut, time, io, mem, net, ai, fs, log
```

11.3　何が効果になるか

構文	効果
フィールドへの代入	`{mut}`
ローカル変数への代入	**効果ではない**
`new C(...)`	`{mem}`
組込み関数の呼び出し	その関数の効果（§22）
`remote(...)` 宛の送信	`{net}`
`now` でのローカル送信	**呼ばれる側の効果を引き継ぐ**
`send` / `future`	引き継がない

`mut` がフィールドへの代入だけなのは意図的で、純粋なアクター（複製してよい）と外部作用を持つアクター（同時に 1 つ）を効果集合だけで区別できるようにするためである。

`now` の伝播は不動点まで反復する。効果は増える一方なので停止する。

11.4　これで何が禁止できるか

```
class Controller {
  method step(s) : string !{io, time} {
    var r = now planner.plan(s) timeout 100 else "";   // plan は !{ai, net}
    reply(r);
  }
}
```

```
[Type error] method Controller.step declares {io, time}
  but its body has {ai, net, io, time} (missing {ai, net})
```

「実時間で動くアクターが、機外のエージェントを同期で呼ぶ」ことが効果検査だけで禁止される。これは設計判断として述べていたものが、型システムの上で機械的に効くようになったということである。

> **既知の穴。** `now` を `future` と `await` に分けて書くと、この検査を**逃れる**。実装は `now` 送信の地点でしか伝播の辺を張らず、`await` では伝播させていないためである。再現は `docs/samples/soundness2_effect_gap.aipl` にある。修正方針（`future` の型に効果を載せる）は第 2 版の報告書に書いてある。

第 III 部
七つの規律

型・効果・期限の上に、七つの規律が載っている。どれも**注釈を書かなくても効く**ように作ってある — 必要な情報は、たいていプログラムの中に既に書かれているからである。

規律	書き方	止めるもの
効果	`!{ai, net}`	実時間側が機外のモデルを待つこと
期限	`timeout` n `else` e	返らない待ち
失敗の型	`timeout` n（`else` 無し）	期限切れの値を正常値として流すこと
返信先の一級性	`replyto` / `answer` / 引数型 `reply`	返信の取りこぼしと二重返信
資源の順序	`acquire` / `release` / `resource_order`	放し忘れ・二重取得・逆順の取得
義務レベル	`@`n（書かなければ推論）	待ちの循環
セッション型	`protocol_define` / `_start` / `_end`	やり取りの順序違反

12　失敗を型に出す — result⟨τ⟩

期限つきの待ちで `else` を書くと、時間切れの値と正常な値が**同じ型**になる。-1 が返ったのか、正しく -1 が計算されたのかをプログラムが問えない。

`else` を書かなければ result⟨τ⟩ が返る。

書き方	型
`now a.m(x) timeout 200 else -1`	`int`
`now a.m(x) timeout 200`	result⟨int⟩

取り出しには `is_ok` / `timed_out` / `value` を使う。

```
var a = now s.fast(42) timeout 200;      // result<int>
if (is_ok(a)) { print("fast ok " ++ value(a, -1)); }
else          { print("fast timed out"); }
```

result⟨τ⟩ は数値としては使えない。確かめずに計算へ流すと止まる。

```
no overload of + matches (result int, int)
```

13　返信先を一級の値にする

`replyto` は、いま処理しているメッセージの返信先を値として返す。型は reply⟨ρ⟩（ρ はそのメソッドの戻り値型）で、`answer(r, v)` で返信する。`reply(v)` はその糖衣にあたる。

引数の型注釈 `reply` を書くと、他のアクターへ渡せる。窓口役が受けて、専門役が**直接**答える形が書ける。

```
class Backend {
```

```
  method handle(r: reply, x) : unit !{} {
    answer(r, x * 2);            // Frontend の依頼者へ直接返す
  }
}
class Frontend {
  var b = new Backend();
  method ask(x) : int !{} {
    send b.handle(replyto, x);   // 返信先を渡して義務を移す。自分は reply しない
  }
}
```

返信先は**線形**に扱われる。ちょうど一度使うのが義務である。

規律	内容
生成	`var r = replyto;` で義務が生まれる
消費	`answer(r, v)` するか、送信の引数として渡すと義務が消える
一度だけ	同じ `r` を二度使ってはならない
残さない	メソッドを抜けるとき義務が残っていてはならない
枝	`if` の二つの枝は同じ状態で合流する

検査はメソッド内で閉じている。返信先を**フィールドに保持する**ことはできない（フィールドは初期化子から型が決まるため）。「送り主を溜めて後で返す」有界バッファのような書き方は、いまのところ核では直接には書けない。

14　資源の順序

効果は**集合**なので、順序を表せない。「取ったら放す」が言えない。そこで `acquire` / `release` の対を本体の中で追う。

```
class Bus {
  var n = 0;
  method read(addr) : int !{mut} {
    acquire("i2c");
    n = n + 1;
    var v = addr * 2;
    release("i2c");              // これを消すと型エラーになる
    reply(v);
  }
}
```

規律は三つ。メソッドを抜けるとき持ち物が残っていてはならない。持っていない資源を放してはならない。二重に取ってはならない。`if` の二つの枝は同じ持ち物で合流し、`while` の本体は持ち物を変えない。

14.1　全体順序 — 逆順の取得を止める

対の検査だけでは足りない。二つのアクターが同じ二つの資源を**逆の順序**で取ると、どちらも対は正しいのに、実行すると互いを待つ。

新しい注釈は要らない。r **を持っているあいだに** s **を取った**という入れ子が、そのまま $r < s$ という主張である。プログラム全体から集め、閉路が無いことを見る。

```
resource order: cache -> db -> cache is circular;
  cache before db (in Reader.run); db before cache (in Writer.run).
  Acquiring in opposite orders deadlocks
```

閉路の各辺が**どこで生まれたか**を持っているので、逆順に取っている二か所がそのまま出る。

推論に任せず `resource_order("bus -> dev")` と書くこともできる。宣言は同じ表に辺として入るだけなので、宣言と実際の取得が食い違えばそれも閉路として出る。推論した順序は `AIOS_SHOW_LEVELS=1` で見える。

```
[resource] bus                  @0
[resource] dev                  @1
```

> 追えるのは資源の名前が**文字列リテラル**の `acquire` だけである。変数に入った名前は静的には分からないので、宣言した順序は実行時にも効かせている（`AIOS_STRICT_RESOURCE=1` で追えなかった場所を知らせる）。

15　義務レベル — 待ちの循環を止める

待ちが返らなくなる形は四つある。循環待ち、返信されない待ち、`select` が来ないメッセージを待つ場合、そして資源を逆順に取る場合である。一つ目に効くのが**義務レベル**である。

各メソッドに番号を付け、**待ちは必ず番号の大きい方へ向かう**ことを要求する。

```
class Leaf { method f(x)  : int @2 !{} { reply(x + 1); } }
class Mid  { var l = new Leaf();
             method g(x)  : int @1 !{} { reply(now l.f(x) timeout 200 else 0); } }
```

書かなくてよい。書かなければ、待ちの辺を見て押し上げる不動点で推論される。明示注釈は固定点として扱い、動かせなければそこが矛盾である。

```
obligation level: B9#g (@2) waits on A9#f (@2);
                  a wait must go to a strictly higher level
obligation levels do not converge; the wait graph has a cycle
```

推論したレベルは `AIOS_SHOW_LEVELS=1` で見える。

15.1　ノードをまたぐ待ち

メッシュでは宛先の実体が別ノードにあり、静的には見えない。そこでノードごとにレベルの**下限**を宣言し、そのノードへの待ちは必ず上へ向かうことを要求する。

```
node_level("rt0", 10);            // 実時間ノードは葉（高いレベル）
```

```
obligation level: Ctl2#go (@7) waits on node rt0 (floor @1);
                  a wait across nodes must go up
```

効果を `node_allow` で国境で照合するのと、まったく同じ形である。

15.2　どこまで保証されるか

待ちの宛先がクラスとして分かるか、宛先ノードのレベルが宣言されている範囲では、デッドロック自由が型で保証される。この主張は紙の上だけではない — Coq/Rocq で、`await` を含んだままの中核について機械検証してある（`coq/AIPLSoundness2.v`、公理なし）。

条件も書いておく。宛先が動的だと見えない。レベルはメソッド単位なので、同じクラスの別インスタンス同士は一律に扱われる。停止性と飢餓は別問題である。

16　セッション型 — やり取りの順序

「開いてから読み、最後に閉じる」という約束は、期限でも送り手の存在でも表せない。AIPL には**実行時**のセッションプロトコルが既にあるので、**同じ宣言**を型検査の側でも読む。新しい宣言は要らない。

```
protocol_define("Pipeline", "f.get -> r.scale -> st.put");
var sid = protocol_start("Pipeline");

var a = now f.get(4)    timeout 200 else 0;
var b = now r.scale(a)  timeout 200 else 0;
var c = now st.put(b)   timeout 200 else 0;
protocol_end(sid);
```

順序を取り違えると、走らせる前に出る。

```
protocol Pipeline: expected r.scale but the program sends st.put here
protocol Pipeline is incomplete at protocol_end; next expected st.put
```

（実際の出力の例）

```
protocol P29: expected b.step2 but the program sends c.step3 here
```

16.1　アクターをまたぐ場合

実際のプログラムでは、手順は一つの並びに収まらない。窓口役をひとつ呼び、残りの手順はその本体の中で進む。

同期の送信（`now` / `aios_now`）は、呼び先が呼び出し側の続きより**先に走り切る**。だから呼び先の送信列を、呼び出しの位置にそのまま差し込んでよい（振る舞い展開）。これで、またいだセッションも一本の並びとして照合できる。

非同期（`send` / `future`）は差し込まない — いつ走るか決まらないからである。呼び先が手順を含んでいたら、完了の判定は実行時に任せる（`AIOS_STRICT_PROTOCOL=1` で知らせる）。

> 手順は**名前**で役を指す。アクターを**引数で渡す**こと。フィールドに持たせると、実行時の呼び名（`c#f`）と静的な呼び名（`f`）が食い違い、手順が役を指せない。

17　`select` の規律

`select` には二つのことを課す。

1. **期限を書く**。`timeout` 節の無い `select` は、来ないメッセージを待ち続ける。
2. **送り手が居ることを確かめる**。`case m(...)` の `m` を、このプログラムの中で誰も送っていなければ、綴り間違いか書き忘れである。

```
select waits for W2.ghost but nothing in this program sends it
select without a timeout waits forever; write `timeout <ms> -> { ... }`
```

いずれも既定は警告である。送信が動的な場合や、外部に公開しているプログラムでは判断できないので、検査そのものを行わない — **言語が世界の全部を見ているわけではない**。

18　メッシュ配備

アクターのソースを他ノードへ送り、**相手先で**構文解析・型検査してから実体化する。効果注釈がコードと一緒に運ばれ、受け入れ側の方針と照合される。

```
node_allow("rt0", "mut, log, time");      // 実時間ノードが受け入れる効果
deploy("rt0", "Servo", "servo1");         // Servo のソースを rt0 へ配って実体化
var d = now remote("rt0", "servo1").step(3) timeout 200 else -1;
```

```
[deploy] Servo -> rt0/servo1 (compiled at destination)
```

`ai` や `net` を持つコードを実時間ノードへ配ると、国境で止まる。効果が、単一ノード内の規律から**ノード間の入国審査**になる。

19　逃げ道（環境変数）

既定はすべて厳しい側に倒してある。緩める向きの環境変数があるが、既定で使うことは想定していない。一覧は §40 にある。

第 IV 部
参考

20　型検査が見るもの

検査	内容
メソッドの存在	ローカル送信先に、その名前のメソッドがあるか
引数の個数と型	呼び出し側の実引数と、本体が要求する型が一致するか
`reply` の型	すべての `reply` が同じ（宣言された）型か
`reply` の回数	高々一度か。注釈があれば全パスでちょうど一度か
select の照合	`case` の arity と引数型が署名と一致するか
公開時の注釈	`web_expose` したアクターのメソッドに注釈があるか
オーバーロード	候補が一意に決まるか（曖昧ならエラー）
効果	宣言した効果が本体の効果を覆っているか。効果名は既知か
期限	`else` 節の型、期限が正であること、期限の有無
失敗の型	`result`$\langle\tau\rangle$ を確かめずに使っていないか
返信先の線形性	`replyto` をちょうど一度使っているか
`reply` の全域性	`now`/`await` で待たれるメソッドが全経路で返すか
資源	`acquire`/`release` の対と、取得順序の閉路
義務レベル	待ちが必ず上へ向かうか。ノード境界も同じ
セッション	送信の順序が `protocol_define` の手順どおりか
select	期限があるか、待つメッセージを誰かが送っているか
未宣言への代入	`var` で宣言していない名前へ代入していないか
配備の境界	配るコードの効果が受け入れ側の方針に収まるか

代表的なメッセージを挙げる。

```
no method foo in actor(C)
type mismatch                                    <- 引数や代入の型が合わない
no overload of + matches (string, int)           <- + は数値専用
ambiguous overload of + for ('a, 'a): candidate return types are float / int
reply type mismatch: this method is declared to reply with int,
  but replies with string here
method m is declared to reply with int, but some execution path does not reply
method m may reply more than once on some path
select: case m binds 1 variable(s) but method m takes 2
actor C is reachable from outside this program, but m has no return type
  annotation; ...
method C.m declares {log, time} but its body has {ai, log, net}
  (missing {ai, net})
unknown effect(s) network in A.m; known effects are mut, time, io, mem, ...
now: the else branch has type string but the call returns int
now: timeout must be positive
now without a deadline waits forever;
  write `now ... timeout <ms> else <expr>`       <- 既定は警告
assignment to undeclared name: cout (write `var cout = ...` to declare it)
no overload of + matches (result int, int)       <- result は確かめてから使う
method Silent4.f is waited on by now/await but does not reply on every path
resource i2c is released without being acquired
method Dev4.use returns while still holding i2c
resource order: cache -> db -> cache is circular; ...
obligation level: B9#g (@2) waits on A9#f (@2);
  a wait must go to a strictly higher level
```

```
obligation levels do not converge; the wait graph has a cycle
protocol P29: expected b.step2 but the program sends c.step3 here
select waits for W2.ghost but nothing in this program sends it   <- 既定は警告
select without a timeout waits forever                            <- 既定は警告
deploy: Servo.step is @3 but node rt0 has floor @10
```

21　外部公開

`web_listen(port)` で HTTP ゲートウェイが起動し、`web_expose(path, actorName)` で公開エンドポイントに別名が付く。

エンドポイント	内容
`GET /`	簡易 UI
`POST /api/send`	`to=<actor>&method=<m>&args=<a,b>` で任意のアクターへ
`POST /api/x/<key>`	`web_expose` した別名へ

`POST /api/send` は**任意のアクターに名前で到達できる**。すなわち境界を開いているのは `web_listen` であって、`web_expose` は別名を付けるだけである。型検査はこれを踏まえ、`web_expose` で名指しされたアクターのメソッド（`init` を除く）は戻り値型注釈が**必須**、`web_listen` だけの場合は**警告**としている。相手からは本体が見えないので、返信の型は宣言でしか伝わらない。

`AIOS_LAX_EXPOSE=1` でエラーを警告に落とせる。

HTTP から渡される引数は文字列を `int` → `float` → `string` の順に解釈して作られる。境界が動的であること自体は注釈では消せないが、**どの型を約束しているか**は宣言できる。

22　組込み関数と、その効果

分類	名前	効果
数学	sin, cos, tan, asin, acos, atan, sqrt, exp, log10, abs, floor, ceil, round	{}
基本	typeof, reply, neg	{}
出力	print	{log}
時間	wait	{time}
配列（読み）	array_get, array_len	{}
配列（割付）	array_empty, array_push, array_set	{mem}
アクター生成	spawn	{mem}
描画 (SDL)	sdl_init, sdl_clear, sdl_line, sdl_erase_line, sdl_present, sdl_line_c	{io}
記憶・タスク	aios_memory_put/get/has/keys, aios_task_create/set/get/info, aios_tasks	{fs}
モデル推論	ai_call, ai_call_with_system, model_generate, gemini_generate, openai_generate	{ai, net}
通信・公開	web_listen, web_expose, aios_now, aios_future, remote_review, remote_review_ja, remote_reviewer_host	{net}
内省	capabilities, capability_prims, aios_kernel, aios_actors, aios_actor_info, aios_actor_methods, aios_mailbox_len, actor_dump	{log}
イベント	aios_emit, aios_events, aios_events_since, aios_event_count	{log}
プロトコル	protocol_define/start/use/current/state/end/events	{log}
サービス	aios_register_service, aios_services, aios_service_actor, aios_service_info	{log}

表に無いプリミティブは効果なしと見なされる。

第 V 部
サンプルプログラム

本節のサンプルは docs/samples/guide/ にある 10 本である。すべて型検査を通り、実行結果も掲載のとおりで、**三つの処理系（OCaml 版・Python 版・JavaScript 版）で出力が一致する**ことを tools/run_all_impls.sh で確かめてある。

より進んだ機能（result⟨τ⟩・返信先の委譲・資源の順序・義務レベル・セッション型・メッシュ配備）のサンプルは docs/samples/paper/ に 20 本ある。本書 §III の例はそこから採った。

23　g1: クラス・フィールド・send・文字列連結

```
// g1: クラス・フィールド・init・メソッド・send・文字列連結
class Greeter {
  var count = 0;              // フィールド。int で初期化
```

```
  method init(name) {          // new のときに自動で呼ばれる
    print("hello, " ++ name); // ++ は文字列連結（両辺を文字列化する）
  }

  method tick() : unit {       // 戻り値注釈。reply しないので unit
    count = count + 1;         // + は数値専用
    print("tick " ++ count);
  }
}

var g = new Greeter("AIPL");
send g.tick();                 // 非同期。返信を待たない
send g.tick();
```

```
hello, AIPL
tick 1
tick 2
```

init は new Greeter("AIPL") のときに自動で呼ばれる。tick は reply しないので unit を宣言してあり、被覆検査は課されない。send は返信を待たないので、2 つの tick は順に処理される。count + 1 は int 同士なので int のままで、表示も 1, 2 と整数になる。

推論された効果は Greeter#init !{log}、Greeter#tick !{log, mut} である。

24 g2: now / future / await と注釈

```
// g2: now / future / await と戻り値型注釈
class Calc {
  // 注釈があると reply はこの型に照合され、全パスで reply することも検査される
  method twice(a) : int {
    reply(a * 2);
  }

  // 注釈を省くと reply から推論される（-> string）
  method label(a) {
    reply("v=" ++ a);
  }
}

var c = new Calc();

var s = now c.twice(21);       // now は式。結果は int
print("twice = " ++ (s + 1)); // int として算術に使える

var f = future c.twice(10);    // future は future int
var v = await f;               // await で取り出す
print("awaited = " ++ v);

print(now c.label(7));         // 推論された string
```

```
twice = 43
awaited = 20
v=7
```

twice は int を宣言しているので now c.twice(21) は int になり、s + 1 という算術に使える。label は注釈を省いているが、reply("v=" ++ a) から戻り値 string が推論される。

このサンプルは期限を書いていないので、型検査時に警告が 3 つ出る。

```
[warn] line 16, col 9: now without a deadline waits forever;
  write `now ... timeout <ms> else <expr>`
```

25　g3: 複数アクターと、アクターを跨いだ now

```
// g3: 複数アクターと、アクターを跨いだ now
class Stock {
  var n = 10;

  method take(k) : int {
    n = n - k;
    if (n < 0) { n = 0; }
    reply(n);
  }
}

class Front {
  // Stock.take の戻り値 int が now の型になり、それが place の string へ繋がる
  method place(k) : string {
    var left = now stock.take(k);
    if (left > 0) {
      reply("ok, left=" ++ left);
    } else {
      reply("sold out");
    }
  }
}

var stock = new Stock();
var front = new Front();

print(now front.place(3));
print(now front.place(99));
```

```
ok, left=7
sold out
```

Stock.take が int を返し、それが Front.place の中の now の型になり、さらに place 自身の string へ繋がる。型はアクター境界を越えて伝播する。

place は string を宣言しているので、if の両方の枝で reply していることが検査される。片方を消すと some execution path does not reply になる。

なお n = n - k で n が int なので k も int に決まり、呼び出し側の place(3) と整合する。ここで place("x") と書けば**引数の**型エラーになる。

効果は両メソッドとも {mut}（フィールド n への代入）で、Front#place は now を通じて Stock#take の{mut} を引き継いでいる。

26 g4: select と reply の帰属

```
// g4: select / case / timeout と、case 本体の reply の帰属
class Worker {
  // 外から届く job メッセージを受け取るメソッド
  method job(k) : string {
    reply("direct:" ++ k);
  }

  // select で job を待つ。case 本体の reply は job への返信になるので、
  // serve 自身は何も reply せず unit のままでよい
  method serve() : unit {
    print("waiting");
    select {
      case job(k) -> {
        print("got " ++ k);
        reply("via select:" ++ k);
      }
      timeout 500 -> {
        print("timed out");
      }
    }
  }
}

var w = new Worker();
send w.serve();
print(now w.job(1));
```

```
direct:1
waiting
timed out
```

case job(k) の本体にある reply は serve ではなく job への返信なので、serve は unit のままでよい。型検査器も case 本体では reply を job の戻り値型に照合する。

実行結果は**この例が抱える設計上の注意**を示している。now w.job(1) が通常のメソッド dispatch で先に処理されたため、serve の select が動いたときには mailbox に job が残っておらず、timeout に落ちた。select で待つメッセージ名は、直接呼ばせない設計にするのが安全である。

27 g5: 外部公開と注釈の必須化

```
// g5: 外部公開すると戻り値型注釈が必須になる
```

```
//     web_listen で境界が開き、POST /api/send で任意のアクターに到達できる。
//     公開先の本体は相手から見えないので、返信の型は宣言でしか伝わらない。
class Echo {
  method say(msg) : string {   // 注釈が無いと型エラーになる
    reply("echo: " ++ msg);
  }
  method note(msg) : unit {
    print("[note] " ++ msg);
  }
}

var echo = new Echo();

web_listen(8080);
web_expose("/echo", "echo");

print("POST /api/x/echo に method=say&args=hi を送ってみてください");
```

このプログラムは型検査を通る。say の注釈を外すと次のようになる。

```
[Type error] line 16, col 1: actor Echo is reachable from outside this
  program, but say has no return type annotation; a remote caller cannot
  see the body, so the reply type has to be declared (method say(...) : T)
```

web_listen(8080) を実行するとゲートウェイが起動して待ち受けに入るので、確認は別の端末から行う。

```
$ curl -s -X POST http://localhost:8080/api/x/echo \
       -d 'method=say&args=hi'
```

28　g6: 効果注釈と伝播

```
// g6: 効果注釈 !{...}
//
//     既存の capability 分類を静的にしたもの。効果は8種:
//       mut   自分の状態を書き換える        net   ノード外への通信
//       time  時刻の取得・待機              ai    モデル推論
//       io    装置への入出力                fs    永続化
//       mem   動的割り付け                  log   観測のみの出力
//
//     注釈を省くと本体から推論される。書いた場合は
//     「本体の効果 ⊆ 宣言した効果」が検査される（多めに宣言するのは可）。
class Counter {
  var n = 0;

  method bump() : int !{mut} {   // フィールドへの代入は mut
    n = n + 1;
    reply(n);
  }
  method peek() : int !{} {      // 効果なし。純粋なので自由に複製できる
    reply(n);
```

```
  }
  method show() : unit !{log} {  // print は log
    print("n = " ++ n);
  }
}

class Planner {
  // ai_call は機外へ出るので ai と net の両方を持つ。
  // これがシグネチャに現れるので、どのアクターが機外へ出るか一目で分かる
  method plan(goal) : string !{ai, net} {
    reply(ai_call("plan: " ++ goal));
  }
}

class ViaNow {
  // ★ now は待つので、呼ばれる側の効果を引き継ぐ。
  //    {ai, net} を宣言しないと型エラーになる
  method run(g) : string !{ai, net} {
    var r = now planner.plan(g);
    reply(r);
  }
}

class ViaSend {
  // ★ send は待たないので引き継がない。効果なしのままでよい
  method fire(g) : unit !{} {
    send planner.plan(g);
  }
}

var planner = new Planner();
var c = new Counter();
var a = new ViaNow();
var b = new ViaSend();

// ai_call は外部API を叩くので、この場では Counter だけ動かす
print(now c.bump());
print(now c.peek());
send c.show();
```

```
1
1
n = 1
```

型検査の出す効果表が、このサンプルの主眼である。

```
[method effects (inferred / declared)]
  Counter#bump !{mut}          <- フィールドへの代入
  Counter#peek !{}             <- 純粋
  Counter#show !{log}          <- print
  Planner#plan !{ai, net}      <- ai_call
  ViaNow#run   !{ai, net}      <- now なので plan の効果を引き継いだ
```

```
  ViaSend#fire !{}            <- send なので引き継がない
```

ViaNow.run から {ai, net} の宣言を外すと declares {} but its body has {ai, net} で落ちる。ViaSend.fire は {} のままでよい。**待つ側だけが呼ばれる側の効果を負う**という規律がここに現れている。

29 g7: 期限付き now / await

```
// g7: 期限付き now / await
//
//     now や await を期限なしで書くと、返信が来なければ永久に待つ。
//     アクターは1台1スレッドで動くので、待っている間そのアクターは
//     他のメッセージを一切処理できない —— 石になる。
//
//     そこで期限と else 節を書けるようにした。
//
//       now t.m(a) timeout <ミリ秒> else <式>
//       await f    timeout <ミリ秒> else <式>
//
//     else 節は本体と同じ型でなければならない（期限切れでも式全体の型は
//     変わらないため）。期限なしの形は当面は警告で、
//     AIOS_STRICT_DEADLINE=1 でエラーになる。
//
//     進行定理の観点では、期限付きの待ちは有限時間で必ず解けるので、
//     期限付きだけを使うプログラムではデッドロック自由が言える。
class Slow {
  method work(k) : int !{time} {
    wait(300);              // 300ms かかる
    reply(k * 2);
  }
  method fast(k) : int !{} {
    reply(k * 2);
  }
}

var s = new Slow();

// 期限内に返る
var a = now s.fast(21) timeout 1000 else 0;
print("fast = " ++ a);

// 期限切れ -> else 節の値になる
var b = now s.work(21) timeout 50 else 0;
print("slow(timed out) = " ++ b);

// future を期限付きで await する
var f = future s.fast(5);
var c = await f timeout 1000 else 0;
print("await = " ++ c);
```

```
fast = 42
slow(timed out) = 0
await = 10
```

work は 300 ms かかるので、期限 50 ms の now は期限切れになって else 節の 0 になる。fast は期限 1000 ms に間に合うので 42 が返る。

このサンプルは AIOS_STRICT_DEADLINE=1 でも通る（すべての待ちに期限が書いてある）。else 0 を else "zero" に変えると

```
[Type error] now: the else branch has type string but the call returns int
```

になる。

30　g8: 等値比較・単項マイナス・真偽値リテラル

```
// g8: 等値比較・単項マイナス・真偽値リテラル
//
//     いずれも字句解析器や評価器には実装があったのに
//     式の文法規則が無く「書けなかった」もの。
//
//       ==  !=          等値。int / float / string / bool のそれぞれで比較できる
//       - e             単項マイナス。neg(e) へ落とす
//       true  false     真偽値リテラル
//
//     優先順位は 等値 < 比較 < ++ < 加減 < 乗除 < 単項マイナス なので、
//     1 + 2 == 3 は (1 + 2) == 3 と読まれる。
class C {
  var flag = false;                    // bool のフィールド
  method set(v) : unit !{mut} { flag = v; }
  method get() : bool !{} { reply(flag); }
  method t() : unit !{log, mut} {
    flag = true;
    if (flag == true)  { print("literal true ok"); }
    if (flag != false) { print("literal false ok"); }
    var b = 1 < 2;
    print("b = " ++ b);
    print("not-eq: " ++ (true != false));
  }
}
var c = new C();
send c.t();
```

```
literal true ok
literal false ok
b = true
not-eq: true
```

いずれも字句解析器と評価器には実装があったのに**式の文法規則が無く**書けなかったもので、前版では「既知の欠落」として挙げていた。flag は false で初期化されているので bool に決まり、set(v) の引数も bool に結線される。

31　g9: アクターを引数に渡す

```
// g9: アクターをメッセージの引数として渡す
//
//     `method m(x: D)` と引数に型注釈を書くと、受け取ったアクターに
//     そのまま send できる。注釈が無いと引数の型が決まらないので
//     `send target is not actor` になる。
//
//     長らく動かなかった書き方で、原因は3つ重なっていた:
//       1. expr_of_value が VActor を "<actor:d>" という文字列に潰していた
//       2. トップレベルの send が引数を評価せず生の式のまま送っていた
//       3. VActor が実体名を持たず、宛先を復元できなかった
class D { method ping() : unit !{log} { print("pong"); } }
class P {
  method viaAnno(x: D) : unit !{log} {
    print("got actor");
    send x.ping();
    print("after send");
  }
}
var d = new D();
var p = new P();
send p.viaAnno(d);
```

```
got actor
after send
pong
```

引数に型注釈 `d: Device` を書くと、その引数のクラスが決まる。`Device` 以外を渡すと型エラーになる。この注釈は義務レベル（§15）の辺も張るので、**アクター型にレベルを載せる必要が無かった**のは、これが理由である。

32　g10: 別々のクラスが同じ名前のフィールドを持つ

```
// g10: 別々のクラスが同じ名前のフィールドを持ってよい
//
//      クラス本体は、そのクラスのフィールドの下で検査される。
//      だから Fork の held と Latch の held は別物である。
//      （OCaml 版はここでフィールドを env に「重ねて」いたため、
//        二つ目のクラスの held への代入が多重定義として弾かれていた。）
class Fork {
  var held = 0;
  method take() : int !{mut} { held = 1; reply(held); }
}
class Latch {
  var held = 0;
  method arm()  : int !{mut} { held = 2; reply(held); }
}
```

```
var f = new Fork();
var l = new Latch();
print("fork  = " ++ (now f.take() timeout 200 else -1));
print("latch = " ++ (now l.arm()  timeout 200 else -1));
```

```
fork  = 1
latch = 2
```

クラス本体は、そのクラスのフィールドの下で検査される。だから `Fork` の `held` と `Latch` の `held` は別物である。

> これは退行防止のサンプルである。OCaml 版はフィールドを型環境に「重ねて」入れていたため、二つ目のクラスの `held` への代入が多重定義として弾かれていた（Python 版と JavaScript 版は正しく通していた）。手元の 586 本のうち 119 本が同名フィールドを持つ二つのクラスを含んでいたが、ほとんどで別のエラーが先に出て隠れていた。

33　負例の一覧

`docs/samples/reply_inference/` には、**通らないこと**を確認するためのサンプルが揃っている。

ファイル	何が起きるか
`s2_missing_reply`	`reply` 忘れの結果を算術に使う
`s3_conflict`	分岐ごとに異なる型を `reply`
`s5_overload_ambiguity`	principal type が無い `a + b`
`s8_annotation_conflict`	宣言と食い違う `reply`
`s9_missing_path`	一部のパスで `reply` しない
`s10_expose_unannotated`	公開アクターに注釈が無い
`s12_service_exposed`	同上（統合サンプルの公開版）
`s14_select_pattern`	`case` の arity が署名と違う
`s15_double_reply`	二重 `reply`
`docs/samples/` 直下	
`soundness2_effect_gap`	`future`+`await` が効果検査を逃れる（既知の穴）

第 VI 部
ビルドと実行

34　処理系のビルド

```
cd src && make thread_repl
```

`dune` でのビルドは `src/lexer.ml` の二重生成で失敗するため、`src/Makefile` を使う。

35　プログラムの実行

REPL コマンドを書いたファイルを `-f` に渡す。

```
cat > run.repl <<'EOF'
load docs/samples/guide/g7_deadline.aipl
compile
EOF
./src/abclrepl_thread -q -f run.repl < /dev/null
```

REPL の主なコマンドは `load` / `compile` / `list` / `send` / `ast` / `pprint` / `script` / `exit` である。

36　型検査だけを走らせる

REPL は SDL に依存するので、型検査専用の小さなドライバを用意している。

```
ocamlfind ocamlc -package unix -thread -linkpkg -w -a -I src -o tc \
  src/location.cmo src/types.cmo src/ast.cmo src/typing_env.cmo \
  src/infer.cmo src/typecheck.cmo src/parser.cmo src/lexer.cmo \
  docs/samples/reply_inference/tc_main.ml

./tc docs/samples/guide/g3_actors.aipl
```

推論・宣言された戻り値型の表、メソッドごとの効果、クラスごとのシグネチャが表示される。

37　型と実行値の差分テスト

型検査が付けた戻り値型と、実行時に実際に `reply` された値の型を突き合わせるハーネスがある。

```
python3 scripts/type_runtime_diff.py abclc/*.aipl docs/samples/guide/*.aipl
```

処理系側は `ABCL_TYPE_TRACE=1` のとき `reply` のたびに `[rtype] Class#method = tag` を出す。この仕組みで、整数演算が `float` を返していた不整合（$\vdash e : \texttt{int}$ なのに $e \Downarrow \texttt{VFloat}$）が実際に見つかった。わざと食い違う負例には`// TYPE_RUNTIME_DIFF: expect-mismatch` と書いておけば既知として扱われる。

型を変更したら必ず回すこと。これが preservation の破れを見つける常設装置である。

38　環境変数

変数	効果
`ABCL_QUIET=1`	型スキーマ等のデバッグ出力を抑制
`ABCL_STRICT_DEADLINE=1`	期限なしの `now` / `await` をエラーにする
`ABCL_LAX_OVERLOAD=1`	曖昧なオーバーロードをエラーではなく警告にする
`ABCL_LAX_EXPOSE=1`	公開アクターの注釈漏れをエラーではなく警告にする
`ABCL_TYPE_TRACE=1`	`reply` のたびに `[rtype]` 行を出す
`ABCL_MODEL_PROVIDER`	AI 呼び出しのプロバイダ選択
`ABCL_AI_PROVIDER`	同上（旧名）
`REMOTE_REVIEWER_HOSTPORT`	`remote_review` の宛先

第 VII 部
付録

39　文法の要約

```
program    ::= decl+

decl       ::= "class" ID "{" field* method+ "}"
             | "var" ID "=" expr ";"
             | "var" ID "=" "new" ID "(" args ")" ";"
             | "send"  target "." ID "(" args ")" ";"
             | "send!" target "." ID "(" args ")" ";"
             | ID "(" args ")" ";"

field      ::= "var"   ID "=" expr ";"
             | "float" ID "=" expr ";"

method     ::= "method" ID "(" params ")" ret? lvl? eff? "{" stmt+ "}"
params     ::= (param ("," param)*)?
param      ::= ID | ID ":" ClassName | ID ":" "reply"
ret        ::= ":" ("int"|"float"|"string"|"bool"|"unit"|"any"|ClassName)
lvl        ::= "@" INT                    -- 義務レベル。書かなければ推論
eff        ::= "!" "{" (ID ("," ID)*)? "}"

target     ::= ID
             | "remote" "(" STRING "," STRING ")"

stmt       ::= ID "=" expr ";"
             | "var" ID "=" expr ";"
             | "var" ID "=" "new" ID "(" args ")" ";"
             | ID "(" args ")" ";"
             | "call" ID "(" args ")" ";"
             | "send"  ("self"|"sender"|target) "." ID "(" args ")" ";"
             | "send!" target "." ID "(" args ")" ";"
             | "if" "(" expr ")" stmt ("else" stmt)?
             | "while" expr "do" stmt
             | "{" stmt* "}"
             | "select" "{" case* timeout_case? "}"

case         ::= "case" ID "(" ids? ")" "->" "{" stmt+ "}"
timeout_case ::= "timeout" INT "->" "{" stmt+ "}"

expr       ::= INT | FLOAT | STRING | "true" | "false" | ID
             | "-" expr
             | expr ("=="|"!="|">"|"<"|">="|"<="|"++"|"+"|"-"|"*"|"/") expr
             | "new" ID "(" args ")"
             | "now" target "." ID "(" args ")" deadline?
             | "future" target "." ID "(" args ")"
             | "await" expr deadline?
             | "replyto"                  -- 返信先を値として取り出す
             | ID "(" args ")"
             | "(" expr ")"

-- else を書くと tau、書かないと result<tau> になる
deadline   ::= "timeout" INT "else" expr
             | "timeout" INT
```

演算子の優先順位は弱い順に、`== !=` ＜ `> < >= <=` ＜ `++` ＜ `+ -` ＜ `* /` ＜ 単項マイナス。二項演算子は**すべて左結合**である（等値も含む）— `1 == 2 == false` は `((1 == 2) == false)` と読まれて `true` になる。単項マイナスは前置なので重ねられる（`- - 3` は `3`）。

40　処理系の環境変数

型検査と実行の挙動を変えるものがある。いずれも `1` を設定して有効にする。既定は「有効にしない」側である。

変数	効果
`AIOS_STRICT_DEADLINE`	期限のない `now` / `await` を**エラー**にする。既定では警告にとどまる。型健全性の定理が成立する範囲（AIPL^{-2}）に留まりたい場合はこれを立てる
`AIOS_LAX_OVERLOAD`	オーバーロードの曖昧さを、エラーではなく「警告して既定候補を選ぶ」**従来動作**に戻す。既定は厳格（エラー）である
`AIOS_LAX_EXPOSE`	公開アクター（§21）の注釈漏れを警告に落とす。既定はエラー
`AIOS_TYPE_TRACE`	`reply` のたびに `[rtype] Class#method = tag` を出す。静的に付いた戻り値型と実際に返った値の型を突き合わせる差分テスト（`scripts/type_runtime_diff.py`）が使う
`AIOS_QUIET`	型検査器と REPL の情報メッセージを抑止する
`AIOS_STRICT_PROTOCOL`	セッションの手順が静的に並べきれない場合の「やり残し」を警告する。既定は黙る（§16）
`AIOS_STRICT_RESOURCE`	名前が文字列リテラルでない `acquire` の場所を知らせる。そこは順序を検査できない（§14）
`AIOS_SHOW_LEVELS`	推論した義務レベルと、資源の全体順序を表示する
`AIOS_LAX_WAIT`	循環待ちの検出をエラーから警告に落とす。既定はエラー
`AIOS_LAX_ACTOR`	アクター型を区別しない（`actor(A)` と `actor(B)` を単一化する）。既定は区別する
`AIOS_LAX_ASSIGN`	未宣言の名前への代入を許す（従来動作）。既定はエラー
`AIOS_MODEL_PROVIDER`	`ai_call` が使うモデル事業者。既定は `gemini`

例えば、期限を必ず書く規律で書きたければ

```
AIOS_STRICT_DEADLINE=1 ./src/abclrepl_thread -q -f run.repl
```

41　よくある落とし穴

- **`now` は式である。**`now x.m();` は構文エラー。必ず `var r = now x.m() timeout 100 else 0;` のように受ける。
- **`send` の宛先は変数**であってクラス名ではない。
- **`now self.m()` は書けない**（自己デッドロックになるため文法で禁止されている）。計算を切り出したいなら別アクターへ。
- **`var x = 0;` に後からアクターを代入できない**（`int` に推論されている）。最初から `var x = new Foo();` で作る。
- **文字列連結は `++`。**`+` は数値専用。
- **フィールドはメソッドより前**に書く。

- **待ちには期限を書く**。書かないと警告が出るし、`AIOS_STRICT_DEADLINE=1` では通らない。
- **効果は申告すると検査される**（gradual）。申告しなければ何も言われない。書けば実体との差を突いてくる。

42 現状の限界

- `future` + `await` に分けて書くと効果伝播を逃れる（§11 の「既知の穴」）。修正には `future` の型に効果を載せる必要がある。
- 型注釈つきの変数宣言 `var x : T = e;` が無い。
- `sender` は `any` で、`send sender.m(...)` はメソッドの存在も型も検査されない。
- リモート送信先（`remote(...)`）の戻り値は `any` のまま。対向ノードのインタフェース記述を読み込む仕組みが無い。
- メソッドの引数は単相で、多相なメソッドは書けない。
- `int` と `float` の間に暗黙変換が無く、両辺が未束縛の `a + b` は注釈が必要。
- 配列リテラル `[...]` は無い。`array_empty` と `array_push` で作る。
- 返信先（`replyto`）を**フィールドに持てない**。引数として渡すことはできる（§13）。フィールドは初期化子から型が決まるので `var w = 0;` は `int` になり、`w = replyto` が型不一致になる。
- 資源の全体順序が追えるのは、名前が**文字列リテラル**の `acquire` だけである（§14）。
- セッションの静的検査が届くのは**同期の呼び先**までである（§16）。
- 停止性は保証されない。デッドロック自由は「詰まらない」であって「必ず終わる」ではない。`while 1 > 0 do {}` は止まらないが、それは待ちではない。

43 前版からの変更点

前版（2026 年 07 月 30 日版）以降の変化を挙げる。

43.1 言語から外したもの

	内容
`become`	**言語から外した**。ABCL/1 に無く、型を守る制限を課すと状態フィールドと分岐で書くのとほぼ同じになり、543 本の実プログラムで一つも使われていなかった。振る舞いの差し替えは状態フィールドと `if` で書く

43.2 足したもの

いずれも**既存のプログラムを壊さない**。注釈は省略でき、省略すれば推論される。

	内容	節
失敗の型	`else` を書かない待ちは `result`⟨τ⟩ を返す	§12
返信先の一級性	`replyto` / `answer` / 引数型 `reply`。線形に扱う	§13
資源の順序	`acquire` / `release` の対と、取得の入れ子から読む全体順序	§14
義務レベル	`@`n。書かなければ推論。ノード境界にも課す	§15
セッション型	実行時のプロトコル宣言を型検査の側でも読む	§16
`select` の規律	期限の義務づけと、送り手の存在検査	§17
メッシュ配備	ソースを送り、相手先で型検査してから実体化	§18

43.3 厳しくなったもの

	内容
未宣言への代入	`var` で宣言していない名前への代入を**弾く**ようになった。以前は黙って新しい変数ができ、フィールド名を打ち間違えても何のエラーも出なかった（`AIOS_LAX_ASSIGN=1` で従来どおり）
アクター型	`actor(A)` と `actor(B)` を区別するようになった（`AIOS_LAX_ACTOR=1` で従来どおり）
`reply` の全域性	`now`/`await` で待たれるメソッドは、戻り値型に関わらず全経路で `reply` することを求める

43.4 拡張子

ソースの拡張子は `.aipl` である（`.abcl` は全廃した）。処理系は拡張子を見ていないので、古いファイルもそのまま読めるが、新しく書くものは `.aipl` にすること。

43.5 核と拡張 — 言語は二層である

実装は三つあるが、**同じ言語ではない**。

層	どこが実装するか	内容
核	三実装すべて	本書が述べている仕様
拡張	Python 実装のみ	記録型・行多相・`where` による精緻化型・CSP チャネル・所有（`pub`/`move`）・タプル・`match`・saga・`scope { future ... }`

本書が仕様として述べるのは核だけである。どちらの層に属するかは `tools/classify_corpus.sh` で機械的に分かる（詳しくは `docs/AIPL_LAYERS.md`）。

44 関連文書

- `docs/aipl_paper5_ja.pdf` — **論文（第 5 版・最新）**。現在の仕様と、ABCL/1 および小林らの型システムからの採否を、理由・メリット・デメリットつきでまとめたもの。
- `coq/AIPLSoundness2.v` — **機械検証（第 2 版）**。Coq/Rocq。`await` を含んだままの中核について、型健全性・型安全性・デッドロック自由を証明してある（公理なし）。三つが同時に成り立つ

最大の構文の議論もヘッダに書いてある。

- `coq/AIPLSoundness.v` — 同第 1 版（デッドロック自由は `await` を除いた断片のみ）。
- `docs/AIPL_LAYERS.md` — **AIPL は二層である**。核（三実装すべて）と拡張（Python 実装のみ）の定義と、`tools/classify_corpus.sh` による分類。
- `docs/aipl_reply_inference_ja.pdf` — `reply` からの戻り値型推論と注釈の設計根拠・測定結果。
- `docs/aipl_vs_abcl1_ja.pdf` — ABCL/1 との比較。
- `docs/aipl_abcl1_features_ja.pdf` — ABCL/1 由来の機能の整理。
- `docs/aipl_kobayashi_synthesis_ja.pdf` — 小林研究の成果を取り込む場合の言語仕様案。
- `docs/ABCLCP_TYPE_SOUNDNESS_REPORT.md` — 型健全性の整理（Markdown）。
- `docs/AIOS_LANGUAGE_DESIGN.md`, `docs/CAPABILITIES.md` — AIOS としての設計と capability の一覧。
- `docs/samples/reply_inference/README.md` — 負例サンプルの意図。