

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 07 月 30 日 17:23 版

対象コミット: 9c25f5f

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 部でサンプルプログラムをソースと実行結果つきで、第 IV 部でビルド・実行・検査の道具立てを、第 V 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)。この三つは互いに噛み合っていて、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」ことが効果検査だけで禁止できる。

本書は前版 (対象コミット 04892cf) を全面的に書き直したものである。前版以降に効果注釈・期限付きの待ち・等値比較・単項マイナス・真偽値リテラルが入り、「既知の欠落」として挙げていた項目のいくつかが消えた。差分だけを知りたい場合は §33 を見ればよい。

目次

第 I 部	はじめに	3
1	AIPL とは	3
2	三つの静的な仕組み	4
2.1	型 — 何を返すか	4
2.2	効果 — 世界に何をするか	4
2.3	期限 — いつまで待つか	4
3	5 分で動かす	5
3.1	処理系のビルド	5
3.2	プログラムを走らせる	5
3.3	型検査だけを走らせる	5
第 II 部	言語ガイド	5

4	プログラムの構成	6
5	字句	7
6	型	7
7	式	8
7.1	優先順位	8
7.2	+ と ++ を混同しない	8
7.3	曖昧なオーバーロード	9
8	文	9
9	アクターと通信	9
9.1	三つの送信	9
9.2	reply	10
9.3	select	10
10	期限付きの待ち	10
10.1	なぜ必要か	10
10.2	規則	11
10.3	型健全性との関係	11
11	効果	11
11.1	8 種類の効果	11
11.2	書き方	11
11.3	何が効果になるか	12
11.4	これで何が禁止できるか	12
12	型検査が見るもの	13
13	外部公開	13
14	組込み関数と、その効果	14
第 III 部 サンプルプログラム		14
15	g1: クラス・フィールド・send・文字列連結	14
16	g2: now / future / await と注釈	15
17	g3: 複数アクターと、アクターを跨いだ now	16
18	g4: select と reply の帰属	17
19	g5: 外部公開と注釈の必須化	17

20	g6: 効果注釈と伝播	18
21	g7: 期限付き <code>now</code> / <code>await</code>	20
22	g8: 等値比較・単項マイナス・真偽値リテラル	21
23	負例の一覧	22
第 IV 部 ビルドと実行		22
24	処理系のビルド	22
25	プログラムの実行	22
26	型検査だけを走らせる	22
27	型と実行値の差分テスト	23
28	環境変数	23
第 V 部 付録		23
29	文法の要約	23
30	処理系の環境変数	24
31	よくある落とし穴	25
32	現状の限界	25
33	前版からの変更点	25
34	関連文書	26

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が続けられる。
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。期限付きの待ちだけを使うプログラムでは**デッドロック自由**が定理として言える (docs/aip1_soundness2_ja.pdf。本書 §10.3 に要約)。

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.abcl
compile
EOF
./src/abclrepl_thread -q -f run.repl < /dev/null
```

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

`-f` に `.abcl` 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 (§26)。ビルドす
ると `./tc file.abcl` で戻り値型・効果・シグネチャの表が出る。

```
$ ./tc docs/samples/guide/g6_effects.abcl
=== g6_effects.abcl ===
[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
<code>class C { ... }</code>	クラス定義
<code>var x = e;</code>	グローバル変数
<code>var x = new C(args);</code>	アクターの生成
<code>send t.m(args);</code>	グローバルな非同期送信
<code>send! t.m(args);</code>	同上（検査を緩めた送信）
<code>f(args);</code>	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, become, call, 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
<code>var x = e;</code>	変数宣言
<code>var x = new C(args);</code>	アクター生成つき宣言
<code>x = e;</code>	代入
<code>f(args);</code> <code>call f(args);</code>	組込み関数の呼び出し
<code>send t.m(args);</code>	非同期送信。返信を待たない
<code>send self.m(args);</code>	自分自身への非同期送信
<code>send sender.m(args);</code>	送信元への非同期送信
<code>send! t.m(args);</code>	検査を緩めた送信
<code>if (e) s if (e) s else s</code>	条件分岐（括弧が要る）
<code>while e do s</code>	繰り返し（括弧は要らない）
<code>{ s ... }</code>	ブロック
<code>become C(args);</code>	振る舞いの置換（効果 <code>{mut}</code> ）
<code>select { ... }</code>	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らない）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;` という型注釈付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
<code>send a.m(x);</code>	待たない	文	負わない
<code>now a.m(x)</code>	返信まで待つ	式（結果は宣言／推論された型）	負う
<code>future a.m(x)</code>	待たない	式（ <code>future T</code> ）	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、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 (§18 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、1ms 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

期限を必須にすると、待ちは「残り時間を減らす簡約」になり、時間が尽きれば `else` 節へ簡約される。その結果**進行定理が二択になり** (終状態か、簡約できるか)、ブロックの枝が消えて**デッドロック自由**が系として言える。第 1 版の報告書が「一般には言えない」と明記した性質である。

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。停止性も依然として言えない。

詳細は `docs/aipl_soundness2_ja.pdf` (型健全性・型安全性の第 2 版) にある。

11 効果

11.1 8 種類の効果

効果	意味	効果	意味
<code>mut</code>	フィールドへの代入	<code>net</code>	ノード外への通信
<code>time</code>	時刻の取得・待機	<code>ai</code>	モデル推論
<code>io</code>	装置への入出力	<code>fs</code>	永続化
<code>mem</code>	動的割り付け	<code>log</code>	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 $\subseteq$ 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
become C(...)	{mut}
new C(...)	{mem}
組込み関数の呼び出し	その関数の効果 (§14)
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.abcl 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。

12 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

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
```

13 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も到達できる。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**。

14 組込み関数と、その効果

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

表に無いプリミティブは効果なしと見なされる。

第 III 部

サンプルプログラム

本節のサンプルは `docs/samples/guide/` にある。すべて型検査を通り、実行結果も掲載のとおりである。

15 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。

16 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>`
```

17 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。

18 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)` が通常の方法 `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。

19 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'
```

20 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。
  // これがシン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.peak());
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。

21 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。

22 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。

23 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	何が起こるか
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 IV 部

ビルドと実行

24 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。

25 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.abcl
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。

26 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.abcl
```

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。

27 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ンネスがある。

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

処理系側は `AIOS_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 ($\vdash e : \text{int}$ なのに $e \Downarrow \text{VFloat}$) 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。

28 環境変数

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

第 V 部

付録

29 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? eff? "{" stmt+ "}"
```

```

ret      ::= ":" ("int"|"float"|"string"|"bool"|"unit"|"any"|ClassName)
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* "}"
          | "become" ID "(" args ")" ";"
          | "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 ")" ("timeout" INT "else" expr)?
          | "future" target "." ID "(" args ")"
          | "await" expr ("timeout" INT "else" expr)?
          | ID "(" args ")"
          | "(" expr ")"

```

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）。

30 処理系の環境変数

型検査と実行の挙動を変えるものが 6 つある。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
<code>AIOS_STRICT_DEADLINE</code>	期限のない <code>now</code> / <code>await</code> 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
<code>AIOS_LAX_OVERLOAD</code>	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
<code>AIOS_LAX_EXPOSE</code>	公開アクター (§13) の注釈漏れを警告に落とす。既定はエラー
<code>AIOS_TYPE_TRACE</code>	<code>reply</code> のたびに <code>[rtype] Class#method = tag</code> 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 (<code>scripts/type_runtime_diff.py</code>) が使う
<code>AIOS_QUIET</code>	型検査器と REPL の情報メッセージを抑止する
<code>AIOS_MODEL_PROVIDER</code>	<code>ai_call</code> が使うモデル事業者。既定は <code>gemini</code>

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
```

31 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。

32 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。
- 型注釈付きの変数宣言 `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。
- `send t.no_such_method()` が通る場合がある。
- 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。
- アクター型が単一化で区別されない（`actor(A)` と `actor(B)` が単一化に成功する）。
- `become C(...)` はコンストラクタの引数しか検査せず、置換後のインタフェースが元と一致するかを見ていない。
- 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。
- 停止性は保証されない。デッドロック自由は「期限付きの待ちだけを使う」場合に限る。

33 前版からの変更点

前版（対象コミット 04892cf）以降の言語の変化は次の 3 点である。いずれも既存のプログラムを壊さない（既定では警告どまり）。

	内容	影響
効果注釈	<code>method m(x) : T !{ai, net}</code> を書けるようになった。省略すれば推論のみ	追加のみ
期限付きの待ち	<code>now ... timeout n else e</code> / <code>await ... timeout n else e</code> 。期限なしは警告 (AIOS_STRICT_DEADLINE=1 でエラー)	追加+警告
式の文法の補完	<code>==</code> <code>!=</code> / 単項マイナス <code>-e</code> / <code>true</code> <code>false</code> が書けるようになった	追加のみ

さらに前版の時点で導入済みだったもの（それ以前から移行が要る点）は、+ から ++ への文字列連結の分離、曖昧なオーバーロードのエラー化、公開アクターへの注釈必須化、仮引数のシグネチャへの結線、二重 `reply` の禁止である。背景と測定結果は docs/aipl_reply_inference_ja.pdf にある。

34 関連文書

- docs/aipl_soundness2_ja.pdf — **型健全性・型安全性の第 2 版（最新）**。効果と期限を含む部分言語 AIPL⁻² の定式化と証明。進行定理が二択になりデッドロック自由が言えること、効果を `future` の型に載せる設計、モデルと実装のずれの一覧。
- docs/aipl_soundness_ja.pdf — 同第 1 版（公開済み。並置してある）。
- 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。