AIPL におけるメソッド戻り値型の推論

— `reply` の引数から型は決まるか —

ABCL^c+ / AIOS 処理系

2026 年 7 月 29 日

概要

AIPL（ABCL^c+）の型健全性・型安全性の検討の中で、「メソッドの戻り値型を宣言すべきだ」という提案が出た。本稿はそれに対する「`reply` の引数から推論できないのか」という問いを出発点とし、(i) 推論は実際に可能であること、(ii) しかし推論だけでは決まらない場面が 6 つあること、(iii) 型健全性の証明にとって本質的なのは推論アルゴリズムではなく宣言という構文であること、を整理する。そのうえで、推論と省略可能な戻り値型注釈 `method m(x) : T` の両方を OCaml 実装に実装し、実測した結果を報告する。推論により `now` / `future` の戻り値型が `any` から具体型へ変わり、既存 63 本のコーパスでは 62 本が判定不変、1 本は誤ったエラーが解消した。注釈は、全実行パスでの `reply` 被覆検査という推論には書けない検査を可能にし、さらに期待型として式の推論へ流れることで**推論そのものを助ける**ことも分かった。さらに、公開アクターのメソッドで注釈を必須とする境界検査を入れた。その過程で、境界を開いているのは `web_expose` ではなく `web_listen` であること、および `select` の 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に返ることが分かり、後者は ρ の帰属の誤りとして修正した。追加機能を一通り使う統合サンプルを書いて実際に処理系で走らせたところ、`apply_binop` が整数どうしの演算でも `VFloat` を返しており、$\vdash e : \texttt{int}$ なのに $e \Downarrow \texttt{VFloat}$ となる preservation の破れが見つかった。型を精密にしたことで初めて観測できたずれであり、整数どうしの `+ - *` を整数のまま返すよう評価器を修正し、除算は型も値も `float` に揃えた。さらに型安全性を上げるため、select パターンのメソッド署名への照合、`reply` の線形性（高々一度）の検査、そして型検査結果と実行時の値を突き合わせる差分テストハーネスを追加した。ハーネスをコーパス 78 本に掛けた結果、残る食い違いは「型検査器自身が曖昧と警告しているもの」1 件だけであることが機械的に確認できた。副次的に、オーバーロード解決 `pick_overload` に潜んでいた 2 つの欠陥（失敗候補の単一化が巻き戻らない／候補の優先順位が登録順の逆）を発見し、これも修正した。最後に、曖昧性の元であった「`+` が数値演算と文字列連結を兼ねる」構造を解消するため、文字列連結を `++` へ分離し、既存コード 38 ファイル・255 箇所を型検査器のエラー位置に従って機械的に移行した。そのうえで、実測に基づいて曖昧な overload をエラーへ格上げし（コーパス 49 本で破綻 0 本）、最大の穴であった「本体の仮引数が署名と繋がっていない」を結線した（57 本のうち影響 2 本、いずれも真陽性）。これにより `method m(a) : int { reply(a + 1); }` を `now c.m("hello")` で呼ぶ誤りが静的に捕まるようになった。

目次

1 問題設定

AIPL のメソッドは値を「返す」のではなく、`reply(v)` によって呼び出し側の future を解決する。

```
class Calc {
```

```
  method twice(a) {
    reply(a * 2);
  }
}
var c = new Calc();
var s = now c.twice(21);
```

ここで s の型は int であってほしい。しかし検討開始時点の型検査器はそう推論していなかった。

1.1 出発点：実装の現状

src/typing_env.ml において reply は次のように、囲んでいるメソッドと一切結び付かない多相プリミティブとしてグローバル環境に置かれていた。

```
add_poly e "reply" (Forall ([(!a).id], TFun ([TVar a], TUnit)))
(* reply : forall 'a. 'a -> unit *)
```

そしてメソッドの型は src/infer.ml の preinfer_all_classes において戻り値 unit 固定で構築されていた（コメントにも「いまは戻り値を unit としておく」とある）。

```
let tfun = Types.TFun (ps', Types.TUnit) in
```

その結果、送信地点は戻り値型を捨てていた。

```
| FutureSend (target, mname, args) -> ...
    (match repr (Types.instantiate sc) with
     | TFun (param_tys, _ret_ty) ->        (* _ret_ty は使われない *)
         List.iter2 (Types.unify ~loc:e.loc) param_tys arg_tys)
    ...;
    TFuture TAny
| NowSend (target, mname, args) ->
    ignore (infer_expr env { e with desc = FutureSend (...) });
    TAny
```

規則として書けば次のとおりで、any が「静的に内容を保証しない境界」として残っていた。

$$\frac{\Gamma(a) = \texttt{actor}(C) \qquad C.m : (\tau_1 \times \cdots \times \tau_n) \to \texttt{unit} \qquad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{now}\ a.m(\vec{e}) : \texttt{any}}\ \textsc{Now-old}$$

2 推論できる部分

結論から言えば推論は可能で、機構としては単純である。メソッド $C.m$ ごとに戻り値型変数 $\rho_{C,m}$ を 1 つ確保し、

- 本体中のすべての reply(v) 地点で $\rho_{C,m}$ と v の型を単一化する、
- すべての now / future 送信地点で $\rho_{C,m}$ を戻り値として返す、

とすればよい。ρ は union-find の link を通じて伝播するので、本体と呼び出し側のどちらが先に検査されても制約は届く。

$$\frac{ret(C,m) = \rho \qquad \Gamma \vdash e : \rho}{\Gamma \vdash_{C.m} \texttt{reply}(e) : \texttt{unit}}\ \textsc{Reply}$$

$$\frac{\Gamma(a) = \texttt{actor}(C) \qquad C.m : (\tau_1 \times \cdots \times \tau_n) \to \rho \qquad \Gamma \vdash e_i : \tau_i \qquad ret(C,m) = \rho}{\Gamma \vdash \texttt{future}\ a.m(\vec{e}) : \texttt{future}\ \rho}\ \textsc{Future}$$

$$\frac{\Gamma \vdash \texttt{future}\ a.m(\vec{e}) : \texttt{future}\ \rho}{\Gamma \vdash \texttt{now}\ a.m(\vec{e}) : \rho}\ \textsc{Now}$$

実装上の要点は 1 つだけある。ρ をメソッドの**型スキームに埋めてはいけない**。$\forall$ に ρ が量化されると `instantiate` が呼び出しごとに ρ を別の変数へ差し替えるため、`reply` 側で付いた制約が呼び出し側へ伝わらなくなる。ρ は単相のまま別表に持つ。

3 推論では決まらないこと

以下の 6 点は、ρ を導入しても推論だけでは決着しない。これが「宣言せよ」という提案の実質的な中身である。

3.1 reply しないパスが黙って通る

どこにも `reply` が無い、あるいは一部の分岐でしか `reply` しない場合、ρ は制約が付かないまま一般化され $\forall\rho.\,\rho$ となる。呼び出し側は「どんな型の値でも返る」と見なすが、実行時にはその `now` は**永久に返らない**。型が嘘をつく。

宣言があれば「全パスがちょうど一度、宣言型で `reply` する」という被覆・線形性検査が書けるが、推論は照合先を持たないためこの義務をそもそも述べられない。

3.2 分岐ごとに異なる型を reply したときの blame

`if p reply(1) else reply("no")` は単一化の失敗にはなるが、エラーは「2 番目に検査された `reply` 地点」に出る。数百行離れた分岐のどちらが誤りかは決められない。宣言があれば各 `reply` が独立に照合され、エラー位置が正しくなる。

3.3 アクター間の相互再帰

$A.m$ が `now b.n()` の結果を `reply` し、$B.n$ が `now a.m()` に依存すると、$\rho_{A,m}$ と $\rho_{B,n}$ が相互依存する。単相であれば単一化で回るが、多相にした瞬間に polymorphic recursion（決定不能）になる。現在の `preinfer_all_classes` による全プログラム事前走査が成立しているのは戻り値が `unit` 固定だったからで、変数にすると呼び出しグラフ上の不動点計算になる。

3.4 リモートアクターと分離コンパイル

`RemoteTarget` は型検査を丸ごとスキップしている。本体が別ノードにあるため推論のしようがない。`web_expose` も同様である。AIPL は分散言語であるから、**宣言は追加の負担ではなく、リモー**

ト境界を型付けする唯一の手段である。推論はソースが手元にある場合しかカバーできず、それは最も安全な場合である。

3.5 `become` と複数実装

振る舞い置換や、同じメッセージに複数クラスが応答する構成では、メッセージの型は「一つの本体の性質」ではなく「複数本体にまたがる契約」である。独立に推論した型の最小上界は principal にならない。

3.6 session protocol への昇格

protocol string を型レベルの session environment へ昇格させるには、$!m(T).?R.\cdots$ の R がインタフェースに書かれていることが前提となる。プロトコルの半分が本体を読まないと分からない状態では昇格できない。

4 型健全性の証明にとっての意味

これが提案の核心である。`now` $a.m(v)$ が飛行中のとき、preservation は構成（configuration / store）の型付け不変条件として「pending future f は型 T を持つ」を述べる必要がある。

T が「callee の `reply` 群を単一化した結果」としてしか定義されていないと、不変条件が callee の本体を参照してしまい、subject reduction が**合成的でなくなる**。callee が一歩簡約するたびに T の意味が変わりうるので、「推論は簡約に対して安定である」という補題が別途必要になる。宣言された T ならば不変条件は $T = ret(C, m)$ という、簡約で動かない構文片で済む。

注意 1. 推論はアルゴリズムの話、宣言はモデルに現れる構文の話である。証明が要求しているのは後者であり、両者は排他ではない。

5 既存言語はどうしているか

「推論を採用しているか」という観点で 4 つ（＋ 1）を調べた結果を表 1 に示す。共通する不変条件は**推論の有無に関わらず、返信の型は必ずどこかに人手で書かれている**ことで、違うのは書く場所だけである。

	本体内の推論	メソッド戻り値	アクター境界
Akka Typed (Scala)	局所推論あり	本体から推論可（再帰は除く）	`ActorRef[R]` として宣言
Erlang	Dialyzer の success typing	静的型なし	`term()`（= any）
Pony	ローカル変数のみ	省略時は推論せず `None`	behaviour は返信を持たない
Encore	`let` のみ	明示必須	宣言から `Fut[T]`
Gleam	HM 系の完全推論	注釈不要	`Subject(msg)` として宣言

表 1　アクター言語における戻り値型の扱い

とくに 2 点が本稿に直結する。

- **Pony**：「戻り値型を省略した関数は `None` を返す」。本体から推論するのではなく `unit` へ defaulting する。§3 第 1 項への対処として、本実装もこの方式を採った。
- **Gleam**：完全な HM 推論を持ちながら、アクターのメッセージ型だけは宣言させる。「推論できる言語ですら境界は書かせる」という最も強い例である。

なお Erlang は `-spec` が強制されない任意注釈であり、`gen_server:call/2` の戻り値は `term()` である。すなわち「宣言する側の前例」ではなく、AIPL の従来状態（`any`）と同じ側に属する。

6　実装

`src/types.ml`（+41 行）と `src/infer.ml` に変更を加えた。

6.1　ρ の表

```
(* "Class#method" -> rho.  Monomorphic on purpose: if rho were quantified
   in the method scheme, instantiate would replace it at every call site
   and the constraint collected at the reply sites would never reach the
   caller. *)
let method_ret_tys : (string, ty) Hashtbl.t = Hashtbl.create 97
let method_ret_key cls m = cls ^ "#" ^ m
let register_method_ret cls m t = Hashtbl.replace method_ret_tys (method_ret_key cls m) t
let lookup_method_ret   cls m   = Hashtbl.find_opt method_ret_tys (method_ret_key cls m)
let reset_method_rets   ()      = Hashtbl.reset method_ret_tys
```

6.2　1 パス目：ρ の確保と defaulting

`preinfer_all_classes` 内の `infer_method` で ρ を確保し、表に登録する。本体に `reply` が構文的に現れないメソッドはその場で `unit` へ落とす。

```
let rho = Types.TVar (Types.fresh_tvar ()) in
Types.register_method_ret c.Ast.cname m.Ast.mname rho;
if not (stmt_has_reply m.Ast.body) then
  Types.unify ~loc:Location.dummy rho Types.TUnit;
(* Keep rho free in the environment so that generalize does not quantify it *)
set_var_scheme env_m "reply"
  (Types.Forall ([], Types.TFun ([rho], Types.TUnit)));
let tfun = Types.TFun (ps', rho) in
```

ρ を `env_m` の `reply` に置いているのは、`generalize (ftv_env env_m)` の自由変数集合に ρ を含めて量化を防ぐためである。

6.3　2 パス目：reply の単相束縛

check_decl のメソッド本体検査環境で、グローバルの $\forall\alpha.\ \alpha \to \texttt{unit}$ を隠すように reply を単相で束縛する。

```
let rho =
  match Types.lookup_method_ret c.Ast.cname m.mname with
  | Some t -> t
  | None   -> TVar (Types.fresh_tvar ())
in
set env_m "reply" (Forall ([], TFun ([rho], TUnit)));
check_stmt env_m m.body
```

reply 専用の型付け関数 check_reply も設けた。pick_overload 任せにすると"no overload of reply matches (string)" という、どの reply と衝突したのか分からない文言になるためである。現在は次のように出る。

```
line 5, col 37: reply type mismatch:
  this method replies with int elsewhere, but with string here
```

6.4　送信地点

FutureSend の戻り値を TFuture TAny から ρ に変え、NowSend は future を剥がして同じ ρ を返す。リモート宛先だけは any のままとした（§3 のとおり、ここは宣言でしか埋まらない）。

7　戻り値型注釈 method m(x) : T

推論を入れたうえで、§3 が届かない部分を埋めるために省略可能な戻り値型注釈を実装した。

```
method add(a, b) : int { reply(a + b); }
method log(msg) : unit { print(msg); }
```

書ける型は int / float / string / bool / unit / any、および大文字始まりのクラス名（actor 型）である。

7.1　字句・構文・AST

- lexer.mll：":" → COLON を追加。この記号は他で使われていない。
- parser.mly：method_decl に注釈つきの規則を追加し、型名を Types.ty へ写す ty_of_name を置いた。float だけは既存の予約語 FLOAT なので別扱いになる。
- ast.ml：method_decl に ret : Types.ty option を追加。構築箇所は parser.mly の 1 箇所だけだった。

注意 2. ast.ml が Types の**値**(string_of_ty_pretty)を参照するようになったため、src/Makefile のリンク順が Wrong link order: Ast depends on Types で落ちるようになった。リンク行で

types.cmo を ast.cmo より前に移して解消している。コンパイル順は元から正しかったが、リンク行だけが逆だった。

7.2 型検査への反映

注釈があれば ρ をその型に単一化する。注釈が無い場合のみ推論に任せ、reply が本体に無ければ unit へ defaulting する。

```
(match m.Ast.ret with
 | Some t -> Types.unify ~loc:Location.dummy rho t
 | None   -> if not (stmt_has_reply m.Ast.body) then
               Types.unify ~loc:Location.dummy rho Types.TUnit);
```

7.3 全パス被覆検査

unit 以外を宣言したメソッドには、**すべての実行パスで reply すること**を課す。これは §3 第 1 項で「推論には書けない」とした検査そのもので、宣言型という照合先があって初めて成立する。

```
let rec replies_on_all_paths (s : Ast.stmt) : bool =
  match s.sdesc with
  | Ast.CallStmt ("reply", _) -> true
  | Ast.Seq ss       -> List.exists replies_on_all_paths ss
  | Ast.If (_, a, b) -> replies_on_all_paths a && replies_on_all_paths b
  | Ast.While (_, _) -> false     (* the body may run zero times *)
  ...
```

while の本体は 0 回実行されうるので false（保守的）とする。式は無条件に評価されるので、式中の reply は「必ず起きる」とみなす。

7.4 双方向型付け：宣言型を式へ流す

注釈を入れただけでは reply(a + b) は通らなかった。a + b の overload 解決が reply より**先に**、しかも a, b が未束縛のまま行われるため、§10 の順位付けで float が選ばれ、そのあと宣言型 int と衝突していた。

そこで check_reply が ρ を**期待型**として引数の推論へ流すようにした（局所的な双方向型付け）。

```
let t = infer_expr ?expected:(expected_of rho) env a in
```

pick_overload は期待型が与えられていれば「戻り値がそれと合う候補」を最優先する。下へ流してよいのは**自身に未確定の型変数を含まない型**だけである（未束縛の ρ をそのまま流すと、候補との照合で ρ 自身が最初の候補の戻り値型に焼き付いてしまう）。

これにより add は (int * int) -> int と、引数側・戻り値側が整合した形で解決する。すなわち**注釈は検査を厳しくするだけでなく、推論を助ける**。

8　リモート境界での注釈必須化

§3 で「宣言でしか埋まらない」とした境界に、実際に注釈を要求する検査を入れた。

8.1　境界を開いているのは web_expose ではない

当初は web_expose("/calc", "calc") で公開されたアクターだけを対象にすればよいと考えていたが、web_gateway.ml を読むとそうではなかった。

```
POST /api/send      -> send to actor by name
    form fields: to=<actorName>&method=<m>&args=<a,b,c>&from=<who>
POST /api/x/<key>   -> send to exposed endpoint
```

この /api/send は、to= で指定された名前のアクターへそのまま Eval_thread.send_message する（handle_send_direct）。つまり**境界を開いているのは web_listen であって、web_expose は公開エンドポイントに別名を付けるだけ**であり、到達可能性を絞ってはいない。

そこで 2 段階で報告することにした。

対象	判定
web_expose で名指しされたアクターのメソッド	型エラー
web_listen があるとき、それ以外のアクター	警告

前者はプログラムが自ら「これが公開インタフェースだ」と宣言しているのでエラーとする（AIOS_LAX_EXPOSE=1 で警告に落とせる）。後者は到達可能ではあるが公開の意図があるとは限らないので警告にとどめる。init は生成時にしか呼ばれないので対象外とした。

リモート送信 now remote(host, actor).m() の相手は別ノードにあり、本体がこのプログラムに無いので、ここでは検査できない。

8.2　既存コードの移行

コーパスで境界を開いているのは web_calc.abcl 1 本だけだった。add に注釈を付けて移行した。型は、注釈が無いときに推論が選んでいた float をそのまま明示して、既存の型付けを変えないようにした。

```
method add(a, b) : float { reply(a + b); }
```

なお実行時の引数は HTTP の文字列を 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。

8.3　発見：select の case 本体の reply は別のメッセージに返る

移行作業中に、ρ の帰属に誤りがあることが分かった。eval_thread.ml は select の case 本体を実行する前に set_current_msg_id m.msg_id で**選択されたメッセージ**の id に差し替える。

```
| Some (m, binds, body_stmt) ->
```

```
    set_current_msg_id m.msg_id;          (* <-- the selected message *)
    List.iter (fun (x,v) -> Hashtbl.replace actor.env x v) binds;
    (try eval_stmt actor body_stmt with _ -> ());
    set_current_msg_id prev_msg_id;
```

したがって

```
method main() : unit {
  select { case add(a, b) -> { reply(a + b); } ... }
}
```

の reply は main ではなく **add への返信**である。当初の実装はこれを main の ρ に単一化していたので誤りだった。修正として

- case 本体の検査では reply を c.pat.meth の ρ に束縛し直す（クラスは self の型から取る）、
- stmt_has_reply は case 本体を数えない（数えると「reply の無いメソッド」の unit defaulting が効かなくなる）、
- replies_on_all_paths で select は false を返す（case が選ばれた実行では囲むメソッドは reply しないまま終わる）

を行った。timeout 本体は囲むメソッドの msg_id のまま走るので、そちらは従来どおり囲むメソッドに属する。

9 実行して分かったこと：評価器が型検査器と食い違っていた

追加した機能を一通り使う統合サンプル（付録 A.11）を書き、型検査だけでなく実際に処理系で走らせたところ、静的な型と実行結果が食い違っていた。

```
stock = 10
ok, left=7.          <-- take : int と宣言したのに小数点が付く
sold out
twice = 42.          <-- twice も推論では int
```

最小再現は次のとおりである（付録 A.13）。

```
class T { method f() : int { reply(1 + 2); } }
var t = new T();
var v = now t.f();
print(v);
```

型検査は T#f -> int で通るが、実行すると 3.、つまり VFloat が返っていた。原因は eval_thread.ml の apply_binop で、整数どうしであっても VFloat を返していたことである。

```
(* --- Int/Float mixed: promote to Float --- *)
| ("+"|"-"|"*"|"/"), v1, v2 when is_number v1 && is_number v2 ->
    let a = as_float_value v1 and b = as_float_value v2 in
    VFloat (match op with "+" -> a +. b | ...)
```

一方 typing_env.ml は + : (int * int) -> int を持っている。すなわち $\vdash e : \texttt{int}$ でありながら $e \Downarrow$ VFloat で、**preservation が成り立っていなかった**。

注意 3. この食い違いは + の型付けが入った時点から存在していたもので、本稿の変更が持ち込んだものではない。しかしメソッドの戻り値型が unit / any に固定されていた間は、型システムが数値について何も約束していなかったため観測できなかった。ρ を導入して戻り値に具体型が付くようになり、さらに注釈で int と**宣言**できるようになったことで、初めて表に出た。型を精密にすると、型システムと実装のずれが可視化されるという例である。

9.1 修正

整数どうしの + - * は整数のまま返すようにし、除算だけは整数除算を行わず float を返す方針に統一した。型側もそれに合わせる。

```
(* eval_thread.ml *)
| ("+"|"-"|"*"), VInt a, VInt b ->
    VInt (match op with "+" -> a + b | "-" -> a - b | "*" -> a * b | _ -> assert false)
(* "/" is not listed here: it falls through to the float case below *)

(* typing_env.ml *)
List.iter add_f3 [ "+"; "-"; "*" ];              (* (int * int) -> int *)
add_mono e "/" (TFun ([TInt; TInt], TFloat));    (* no integer division *)
```

/ を int に型付けしたままにすると、7 / 2 の型が int で値が 3.5 になり、逆向きの食い違いが生まれる。また整数除算に変えると既存プログラムの数値が変わってしまう。どちらも避けるため、/ は型も値も float に揃えた。

修正後は四則すべてで静的な型と実行結果が一致する。

式	型	修正前の値	修正後の値
7 + 2	int	9.	9
7 - 2	int	5.	5
7 * 2	int	14.	14
7 / 2	float	3.5	3.5

統合サンプルの出力も揃った。

```
stock = 10
ok, left=7
sold out
twice = 42
[stock] done
```

9.2 残る食い違いと、その境界

a + b の両辺が未束縛の型変数のときは、§10 のとおり principal type が存在しない。型検査は float を選んで警告を出すが、実行時に整数が渡れば VInt が返るので、ここだけは依然として食い違う。

```
class A { method add(a, b) { reply(a + b); } }
```

```
...
[warn] ambiguous overload of + for ('a, 'a): ... (picked float)
  A#add -> float          <-- 静的
3                         <-- 実行結果は VInt
```

ただしこの残余は**型検査器が自ら「曖昧である」と警告しているプログラムに限られる**。戻り値型を注釈すれば期待型が overload 解決へ流れて `int` に確定し、実行結果とも一致する（付録 A.7）。すなわち「principal type が無いところでは実装と型が揃わないことがある」という形に問題が閉じ込められており、これは注釈を書く動機として §3 が挙げたものと同じである。

注意 4. 本稿の退行試験（§14）はコーパス 63 本すべてで判定が変わらないことを確認しているが、これは**型検査の**判定である。今回の `apply_binop` の変更は**実行時の値**を変える（整数演算の結果が `7.` から `7` になる）。表示や文字列連結の見た目が変わるので、その点は既存プログラムに影響する。

10　副産物：`pick_overload` の 2 つの欠陥

ρ を可視化した結果、`method add(a,b) { reply(a+b); }` の推論結果が `(int * int) -> string` という自己矛盾したシグネチャになることが判明した。原因は本変更ではなく、既存の `pick_overload` にあった。

(1) **失敗候補の単一化が巻き戻らない**　`List.for_all2` は途中で `false` になると打ち切るため、それまでに結んだ束縛（例：第 1 引数 := `string`）が残ったまま次の候補へ進む。候補を試すたびに引数の型が汚染される。

(2) **候補の優先順位が登録順の逆**　`typing_env` は overload を `sch :: prev` で積むので、リストの先頭は**最後に**登録されたものになる。`+` では `('a * string) -> string` が先頭であり、引数が両方とも未束縛の型変数のときは必ずこれが最初に一致する。`a + b` が問答無用で `string` に潰れていたのはこれが理由である。

修正は次の 3 点からなる。

1. 各トライアルの前後で型変数の link をスナップショット・復元し、候補試行が副作用を残さないようにする。link は `repr` が `cell := {!cell with link}`、`prune` が `(!cell).link <- ...` と 2 通りの書き換えをするため、レコードを共有したまま保存すると復元にならない。link を**値として**保存し、新しいレコードを作り直して差し替える。
2. 候補を「引数型が具体的なもの優先 → 引数側の型変数を束縛しないもの優先 → 登録順」で順位付けする。
3. 最上位に戻り値型の異なる候補が複数残る場合は真に曖昧なので報告する。既定では警告、`AIOS_STRICT_OVERLOAD=1` でエラーになる。

順位付けの最優先キーは §7.4 の期待型との照合であり、その次が「引数型が具体的」、次が「引数側変数を束縛しない」、最後が登録順である。

```
[warn] line 10, col 13: ambiguous overload of + for ('a, 'a):
       candidate return types are float / int (picked float)
```

注意 5. $a + b$ において a, b が未束縛のとき、この overload 集合のもとでは principal type が存在しない。`reply` の型が推論だけでは決まりきらないのと同じ現象であり、注釈を要求する動機はここでも共通している。

11　型安全性をさらに上げる：3 つの追加

ここまでで戻り値については型が付くようになった。では他はどうか、を実測した。小さなプログラムを型検査器に通して、通ってはいけないものが通らないかを確かめた結果が表 2 である。

#	穴	確認に使ったプログラム	結果
1	仮引数が署名と繋がっていない	`m(a) : string { reply(a + "x"); }` に `now c.m(1)`	通る
2	select パターンが署名と照合されない	`method m(a,b)` に対し `case m(x)`	通る
3	`reply` の線形性がない	`m() : int { reply(1); reply(2); }`	通る
4	`sender` 宛の送信が無検査	`send sender.no_such_method(1);`	通る
5	actor 型が単一化で区別されない	`unify` に `TActor(_,_), TActor(_,_) -> ()`	潜在

表 2　型検査を通ってしまうもの（実測）

1 が最も深刻である。型検査器は引数に精密な型を**付けて**いるが、本体とは**照合していない**。`preinfer` が作ったスキームの引数型変数と、`check_decl` が本体用に振る fresh 変数が別物だからで、§15 の限界 2 に挙げた「本体と署名が別世界」の実害である。ただしこれを直すとクラス本体が呼び出しより先に検査される順序のせいで既存コードが大量に落ちるため、§10 の overload の曖昧性を先に解消しておく必要がある。本節では影響範囲の小さい 2, 3 と、そもそもこの種のずれを自動で見つける仕組みを入れた。

11.1　select パターンをメソッド署名に照合する

select の case は、同じアクター宛の同名メッセージを受け取る。これを同名メソッドの署名に照合し、arity と引数型を単一化するようにした。パターン変数には fresh な型変数ではなく**署名の引数型**を束縛する。

arity が合わない case は、実行時には `match_callstmt` が「一致しない」と判断するので**永久に選ばれず timeout に落ち続ける**。エラーにならないので書いた側は気づかない。付録 A.14 参照。

11.2　`reply` の線形性

`resolve_future` は 2 度目の呼び出しでも黙って `fresult` を上書きするだけで、待ち手はすでに 1 度目の値を受け取っている。つまり 2 度目の `reply` は捨てられるが、エラーにはならない。

そこで reply 回数の構文的な上界 `max_replies` を定義し、2 以上になる本体を拒否するようにした。分岐は両側の `max`、逐次は和、**ループ本体に `reply` があれば 2 回以上**とみなす。`select` の case 本体は別メッセージ宛なので囲むメソッドには数えず、case ごとに独立に検査する。

注釈のあるメソッドには既に「全パスで少なくとも一度」（下界）を課しているので、上界と合わせて**ちょうど一度**になる。付録 A.15 参照。

11.3　型と実行値の差分テスト

§9 の食い違いは、型検査の結果と実行結果を突き合わせる仕組みがあれば即座に出ていた。そこでこれを常設した。

処理系側は `AIOS_TYPE_TRACE=1` のとき、`reply` のたびに「どのメソッドが何型の値で返したか」を出す。

```
[rtype] Stock#take = int
[rtype] Order#place = string
```

メソッド名は実行文脈から取る。`exec_context` に `ctx_method_name` を足し、`actor_loop` のディスパッチと `select` の case 実行の両方で設定する（後者は §8.3 のとおり帰属先が変わるため）。

`scripts/type_runtime_diff.py` が、型検査器の出す `Class#method -> 型` と、この `[rtype]` 行を突き合わせる。型変数のまま（`'a`）や `any` は「型システムが何も約束していない」ので判定しない。わざと食い違う例には

```
// TYPE_RUNTIME_DIFF: expect-mismatch
```

と書いておくと既知として数え、終了コードを 1 にしない。

12　文字列連結を ++ に分離する

§10 の曖昧性の元は、`+` が数値演算と文字列連結を兼ねていたことである。`typing_env` には 4 つの候補があった。

```
(float * float) -> float
(int   * int)   -> int
(string * 'a)   -> string
('a * string)   -> string
```

両辺が未束縛の `a + b` ではこの 4 つが**どれも一致する**。しかも候補列は登録順の逆なので先頭は `('a * string) -> string` で、これが必ず選ばれて引数まで `string` に焼き付いていた。

12.1　変更

文字列連結を `++` という別の演算子に分けた。

- `lexer.mll`：`"++"` → `PLUSPLUS`。ocamllex は最長一致なので `"+"` と衝突しない。
- `parser.mly`：`%left PLUSPLUS` を `PLUS MINUS` より**前**に置き、最も弱い結合とした。`"x=" ++ a + b` が `"x=" ++ (a + b)` と読まれる。
- `typing_env.ml`：`+` から文字列 2 候補を外し、`++` : $\forall\alpha\beta.\ (\alpha \times \beta) \to$ `string` を 1 つだけ登録。候補が 1 つなので曖昧になりようがない。
- `infer.ml`：`Binop` にあった「`+` の片側が `string` なら `string`」という特例を削除。`+` は純粋な数値演算になった。
- `eval_thread.ml`：`apply_binop` の `+` の文字列ケースを削除し、`++` は両辺を文字列化して連結する。

```
"n=" ++ 42   ->  "n=42"       1 ++ 2  ->  "12"
"a"  ++ "b"  ->  "ab"         1 +  2  ->  3
"a"  +  "b"  ->  no overload of + matches (string, string)
```

12.2　既存コードの移行

コーパスとサンプルで文字列連結に + を使っている箇所は多い。テキストの見た目で数値の + と区別するのは危ういので、**型検査器のエラー位置に従って機械的に置き換えた**。Binop の位置は mk_expr1 2、すなわち演算子トークンそのものを指すので、no overload of + matches が出た (line, col) の 1 文字を ++ にして再検査、を繰り返せばよい。純粋な数値の + はそもそもエラーにならないので触られない。

結果は **38 ファイル / 255 箇所**である。移行前後で型検査の判定はコーパス 63 本すべて一致した。

注意 6. この移行で 2 つ踏んだ。ひとつは**列がバイトオフセット**であること（pos_cnum - pos_bol + 1）。日本語を含む行では文字インデックスとずれるので、置換はバイト列で行う必要がある。もうひとつは、この方法が**文字列連結以外の理由で失敗している + も書き換えてしまう**こと。実際サンプル s2_missing_reply.abcl の print(x + 1) は「x が unit だから」失敗する意図的な負例なのに ++ にされてしまった。意図的にエラーになるコードは手で確認が要る。

12.3　どこまで消えたか

string 由来の曖昧性は完全に消えた。引数が string に焼き付く事故も起きない。一方、**数値の int / float の曖昧性は残っている**。

```
[warn] ambiguous overload of + for ('a, 'a):
       candidate return types are float / int (picked float)
```

候補が 4 つから 2 つに減り、警告が出るのはコーパス 63 本では 0 本、サンプルでは意図的な負例 2 本だけになった。完全に消すには + と +. のように数値側も分ける必要があり、これは int と float をどう扱うかという別の設計判断になる。

13　仮引数の結線と、曖昧性の厳格化

§11 の表 2 で「最も深刻」とした穴 1（仮引数が署名と繋がっていない）に手を入れた。前節までの整理で条件が揃ったためである。

13.1　測って順序を決める

前節では「穴 1 を直すとクラス本体が呼び出しより先に検査される順序のせいで既存コードが大量に落ちる」と見積もっていた。実際に試作して測るとそうではなかった。

変更	影響	
曖昧 overload をエラーにする	OK だった 49 本のうち 0 本が破綻	ほぼ無料
仮引数を署名に結線する	OK だった 57 本のうち 2 本	予想より安い

しかも落ちる 2 本はどちらも真陽性だった。

- `abclc/Hello.abcl` — フィールドが `float count = 0.;` で `init(n) { count = n; }` なので本体は n : `float` を要求する。しかし `new Hello(5)` は `int` を渡していた。**本物の int/float 不整合**である（`new Hello(5.)` が正しい）。
- `s5_overload_ambiguity.abcl` — 本体が `float`、呼び出しが `int`。principal type が無いことを示す負例なので当然。

見積りが外れたのは §12 の `++` 分離が効いたためである。文字列連結の候補が消えて引数が `string` に焼き付かなくなったので、本体で決まる型と呼び出し側の型が衝突しにくくなっていた。**順序を先に整えたことが、後の変更を安くした**。

13.2 曖昧 overload をエラーにする

これまで曖昧な overload は警告を出して既定候補（`float`）を選んでいた。しかしそれは静的に `float`、実行時に `int` という食い違いを生んでいた（§9）。型検査器が自ら「曖昧」と言っているのだから黙って選ぶべきではない。実測でコストが 0 だったので既定でエラーにした。`AIOS_LAX_OVERLOAD=1` で従来動作に戻せる。

13.3 仮引数を署名に結線する

`check_decl` で仮引数に fresh な型変数を振るのをやめ、`preinfer` が作ったスキームの引数型を束縛する。

```
let fresh_params () =
  List.iter (fun p -> set env_m p (Forall ([], TVar (Types.fresh_tvar ())))) m.params
in
(match Types.lookup_class_method_scheme c.Ast.cname m.mname with
 | Some sc ->
     (match repr (Types.instantiate sc) with
      | TFun (ps, _) when List.length ps = List.length m.params ->
          List.iter2 (fun p t -> set env_m p (Forall ([], t))) m.params ps
      | _ -> fresh_params ())
 | None -> fresh_params ());
```

スキームの引数型変数は generalize されていない（`preinfer` が `env_m` に置くので `ftv_env` に入る）ので、`instantiate` してもそのまま残り、すべての送信地点と共有される。したがってここで束縛すれば、本体と呼び出し側の双方から同じ変数へ制約が集まる。

これで穴 1 が閉じた。

```
class C { method m(a) : int { reply(a + 1); } }
var r = now c.m("hello");
```

```
変更前: OK                                   <-- 引数の型検査が効いていない
変更後: line 3, col 9: type mismatch
```

注意 7. 2 つの変更は互いを補強する。`AIOS_LAX_OVERLOAD=1` にしても `s5` は通らない。曖昧性が `float` に既定された結果が呼び出し側の `int` と衝突するからで、結線が入った以上「黙って選ぶ」だけでは矛盾を隠せなくなっている。

注意 8. メソッドの引数は結線の前後いずれも**単相**である。スキームの引数型変数は generalize されていないため全呼び出し地点で共有され、同じメソッドを `int` と `string` で呼ぶと衝突する。多相にするにはスキームを一般化する必要があるが、それは ρ が量化されてしまう問題（§2 の要点）を再燃させる。証明との相性を考えると単相のままが妥当である。

14 実験

14.1 サンプル

6 本のサンプルで、パッチ前（`git HEAD`）と後の判定を比較した。

サンプル	パッチ前	パッチ後
s1 基本	OK	OK。 `twice:(int)->int`, `greet:(string)->string`, `log:('a)->unit`
s4 アクター跨ぎ	OK	OK。`Store#get->int` が `Front` の `now` を経て `Front#fetch->string` へ伝播
s6 now の結果を算術に使う	`no overload of + matches (any, int)`	**OK**
s3 分岐で異なる型を reply	**OK（見逃し）**	`reply type mismatch: ... int ... string`
s2 reply 書き忘れ	`(any, int)`	`(unit, int)`（原因が読める文言に）
s5 overload の曖昧性	`-> string`（黙って）	`-> float` ＋ ambiguous 警告

表 3　サンプルにおけるパッチ前後の判定

s6 が「正しいプログラムが通るようになった」、s3 が「誤ったプログラムが落ちるようになった」であり、両方向に効いている。

14.2 退行試験

`abclc/*.abcl` 63 本について、`git HEAD` 版と修正版の型検査結果を比較した。

判定が一致	62 / 63
判定が変化	1 / 63
曖昧警告が出たファイル	2

14.3　型と実行値の差分テスト

§11.3 のハーネスを、コーパスとサンプルの計 78 本に掛けた。

```
$ python3 scripts/type_runtime_diff.py --timeout 12 \
    abclc/*.abcl docs/samples/reply_inference/s*.abcl
対象 78 本 / 突き合わせできた 32 本 / 照合したペア 90 件
  （型変数・any のため判定せず 6 件）

想定外の食い違いなし
```

§13 で曖昧な overload をエラーへ格上げしたことで、最後まで残っていた `s5` の食い違いも消えた。**静的には `float`、実行時は `int`** という状態そのものが型検査を通らなくなったためである。`expect-mismatch` 宣言も不要になり、いまは既知・未知いずれの食い違いも報告されない。

突き合わせできたのが 32 本にとどまるのは、残りが API キーやネットワーク、SDL を必要とするか、待ち受けに入って `reply` を一度も実行しないためである。判定できないものは失敗にしていない。

この節を最初に書いた時点では `s5` が唯一の食い違いだった。「残る食い違いは型検査器自身が曖昧と警告しているものに一致する」という性質が機械的に確認でき、それが §13 で「ならば警告ではなくエラーにすべきだ」という判断の根拠になった。`abclc/now_future.abcl` も同じ形だったので `: int` を注釈して静的・動的を揃えてある。

14.4　型検査の退行

変化した 1 本 `RotateThreeLines.abcl` は、

```
x1 = cx - dx + x;   // cx, dx : float, x : method parameter
```

において、旧実装が `('a * string) -> string` を先に選んで `x` を `string` に固定してしまい、`line 19: type mismatch` という**誤ったエラー**を出していた。修正後は具体的な `(float * float) -> float` が選ばれ、`x` は正しく `float` に束縛されて型検査を通る。すなわち退行ではなく改善である。

曖昧警告が出た 2 本（`now_future.abcl`, `web_calc.abcl`）はいずれも `method add(a,b) { reply(a+b); }` の形であり、真に principal type を持たない箇所である。

15　残る限界

1. **戻り値は単相である。** ρ を全呼び出し地点で共有するため、戻り値に多相性は持たせられない。多相にすると §3 の相互再帰が決定不能側へ落ちる。
2. **本体と署名がなお別世界である。** メソッド本体の仮引数は `check_decl` が振る fresh 変数であり、`preinfer` が作ったスキームの引数型変数とは繋がっていない。そのため s5 のように `(int * int) -> float` という、引数側が呼び出し地点由来・戻り値側が本体由来の混成シグネチャが表示されうる。結線は素朴に行うとクラス本体が呼び出しより先に検査される順序のせいで誤検出が増えるため、本変更では触っていない。
3. **リモート送信先は `any` のままである。** これは実装の手抜きではなく、宣言なしには原理的に埋まらない。送信**する**側は相手の本体を持たないので、対向ノードのインタフェース記述を読み

込む仕組みが要る。

4. **principal type が無い式では、型と実行結果が揃わないことがある**（§9）。`apply_binop` の食い違いは修正したが、両辺が未束縛の `a + b` は型検査が `float` を選ぶ一方実行時に `VInt` が返りうる。型検査器はこれを ambiguous と警告しており、注釈を書けば解消する。

16　結論と次の一歩

`reply` の引数からメソッドの戻り値型を推論することは可能であり、実装も 200 行程度で済んだ。効果も実測できた。しかしそれは提案を否定するものではない。

推論が届かないのは (i) `reply` を書き忘れたパス、(ii) 分岐間の blame、(iii) アクター間の相互再帰、(iv) リモート・分離コンパイル、(v) `become` と複数実装、(vi) session protocol への昇格であり、いずれも AIPL が分散アクター言語であることに由来する。そして型健全性の証明が要求しているのは、推論アルゴリズムではなく簡約で動かない構文としての宣言である。

したがって両者は排他ではなく役割分担である。本稿ではこれを実装した。

(済) AST に省略可能な `method m(x) : T` を追加した（§7）。

(済) 注釈がある場合：本体環境に $\texttt{reply} : T \to \texttt{unit}$ を束縛し、全パス被覆検査を行い、送信地点で T を使う。さらに T を期待型として式の推論へ流す（§7.4）。

(済) 注釈が無い場合：推論を走らせる。ただし制約の付かない ρ は一般化せず `unit` へ defaulting する（Pony 方式）。ここだけは一般化してはならない。`send`（fire-and-forget）と `now`（ask）の区別がこの一点に乗る。

(済) **公開するメソッドで注釈を必須にする**（§8）。境界を開いているのは `web_expose` ではなく `web_listen` であることが実装を読んで分かったので、名指しされたアクターはエラー、それ以外は警告の 2 段階とした。引数が文字列リテラルでない `web_expose` は「静的に確認できない」と報告する。

(未) 推論結果のシグネチャを出力し、利用者が貼り付けられるようにする。

実装して分かった副次的な帰結として、**注釈は検査を厳しくするだけでなく推論を助ける**。`reply(a + b)` は注釈が無ければ principal type を持たず `float` へ既定されるが、`: int` と書くだけで期待型が overload 解決へ流れ、`(int * int) -> int` と整合した形に決まる（§7.4、付録 A.7）。宣言は制約であると同時に情報でもある。

いずれの設計でもクラスのメソッド表には結局インタフェースが生まれる。論点は**存在するかどうか**ではなく**誰が書くか**であり、リモートと証明の都合から「人間が書く」側に倒すのが妥当である。その先に、protocol string を型レベルの session environment へ昇格させる道が開ける。

付録 A　サンプル全文と実行結果

本節のサンプルは `docs/samples/reply_inference/` に置いてある。型検査だけを走らせる小さなドライバ `tc_main.ml`（§ 付録 B）を用いて実行した。

```
$ cd src && make thread_repl
$ 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/reply_inference/s1_ok.abcl
```

A.1 s1_ok.abcl — reply からの推論

```
// (1) reply から戻り値型が推論されること
class Calc {
  // reply が int
  method twice(a) {
    reply(a * 2);
  }
}

class Greeter {
  // reply が string
  method greet(name) {
    reply("hello, " ++ name);
  }
  // reply が無い = tell 型のメソッド。unit へ defaulting されるべき
  method log(msg) {
    print(msg);
  }
}

var calc = new Calc();
var g = new Greeter();

var s = now calc.twice(21);
print("twice = " ++ s);

var f = future g.greet("AIPL");
var msg = await f;
print(msg);
```

```
[OK] type check passed
[method return types (inferred from reply)]
  Calc#twice -> int
  Greeter#greet -> string
  Greeter#log -> unit
[class_method_schemes]
class Greeter
  greet : (string) -> string
  log : ('a) -> unit
class Calc
  twice : (int) -> int
```

`twice` は `int`、`greet` は `string` と推論され、`reply` を持たない `log` は `unit` へ defaulting されている。

A.2 s2_missing_reply.abcl — reply の書き忘れ

```
// (2) reply を書き忘れたメソッドの結果を now で使う
//     従来: now の型が any なので素通り → 実行時に永久に返らない
//     今回: bump の戻り値が unit に defaulting され、静的エラーになる
class Counter {
  method bump(n) {
    print("bumped");
  }
}

var c = new Counter();
var x = now c.bump(1);
print(x + 1);
```

```
[Type error] line 12, col 9: no overload of + matches (unit, int)
[method return types (inferred from reply)]
  Counter#bump -> unit
```

パッチ前は `(any, int)` というエラーで、「そもそも `bump` が値を返さない」という原因が読み取れなかった。現在は `unit` と出るので、`reply` の欠落が直ちに分かる。

A.3 s3_conflict.abcl — 分岐ごとに異なる型を reply

```
// (3) 分岐ごとに違う型を reply する
//     ρ が2つの reply 地点を結ぶので、2つ目の reply で衝突が検出される
class Router {
  method route(k) {
    if (k > 0) { reply(1); } else { reply("negative"); }
  }
}

var r = new Router();
var v = now r.route(5);
```

```
[Type error] line 5, col 37: reply type mismatch:
  this method replies with int elsewhere, but with string here
[method return types (inferred from reply)]
  Router#route -> int
```

パッチ前はこのプログラムが**型検査を通っていた**。ρ が 2 つの `reply` 地点を結ぶことで検出できるようになった。ただし §3 のとおり、エラーが指すのは「2 番目に検査された `reply`」であって、どちらが誤りかは決められない。

A.4 s4_chain.abcl — アクターを跨いだ伝播

```
// (4) アクターをまたいで ρ が伝播すること
//     Store.get の reply -> Front の now の型 -> Front.fetch の reply
class Store {
  method get(k) {
    reply(k * 100);
  }
}

class Front {
  method fetch(k) {
    var v = now store.get(k);
    reply("value=" ++ v);
  }
}

var store = new Store();
var front = new Front();

var out = now front.fetch(3);
print(out);
```

```
[OK] type check passed
[method return types (inferred from reply)]
  Front#fetch -> string
  Store#get -> int
[class_method_schemes]
class Store
  get : ('a) -> int
class Front
  fetch : (int) -> string
```

Store.get の reply から得た int が、Front.fetch 内の now の型となり、さらに Front.fetch 自身の reply を経て string に至る。ρ が union-find 経由でアクター境界を越えて伝播していることが確認できる。

A.5 s5_overload_ambiguity.abcl — principal type が無い例

```
// (5) principal type が無い式は拒否される
//
//     a も b も未束縛の型変数なので、+ の候補がどれも同じ資格で一致する。
//
//     かつては + が数値2種に加えて文字列連結2種
//       (string * 'a) -> string / ('a * string) -> string
//     も持っていた。typing_env は overload を prepend で積むためリストの
//     先頭が最後に登録されたものになり、a + b は問答無用で string に
```

```
//     潰され、b も string に焼き付いていた。
//     RotateThreeLines.abcl が誤ってエラーになっていたのもこれが原因。
//
//     文字列連結を ++ に分けたので string の候補は消えたが、
//     (float * float) -> float と (int * int) -> int が残り、
//     principal type はやはり無い。
//
//     以前はここで警告を出して float を既定に選んでいたが、静的には float、
//     実行時には int が返るという食い違いを生んでいた（型検査器が自ら
//     「曖昧」と言っているのだから、黙って選ぶべきではない)。
//     コーパス 49 本で実測して 1 本も落ちなかったので、いまはエラーにしている。
//     AIOS_LAX_OVERLOAD=1 で従来の警告動作に戻せる。
//
//     直し方は戻り値型を注釈すること。期待型が overload 解決へ流れて
//     一意に決まる（s7 を参照)。
class Calc {
  method add(a, b) {
    reply(a + b);
  }
}

var calc = new Calc();
var s = now calc.add(1, 2);
```

```
[warn] line 10, col 13: ambiguous overload of + for ('a, 'a):
       candidate return types are float / int (picked float)
[OK] type check passed
[method return types (inferred from reply)]
  Calc#add -> float
[class_method_schemes]
class Calc
  add : (int * int) -> float
```

AIOS_STRICT_OVERLOAD=1 を与えると警告ではなくエラーになる。

```
TYPE_ERROR: line 10, col 13: ambiguous overload of + for ('a, 'a):
  candidate return types are float / int
```

表示されている (int * int) -> float は、引数側が呼び出し地点由来、戻り値側が本体由来の混成である（§15 の限界 2)。

A.6 s6_usable_result.abcl — 通るようになった正しいプログラム

```
// (6) 正しいプログラムが通るようになる例
//     パッチ前: now の結果は any なので、算術に使った時点で
//               "no overload of + matches (any, int)" で落ちていた。
//     パッチ後: reply(a * 2) から int と分かるので通る。
class Calc {
  method twice(a) {
```

```
    reply(a * 2);
  }
}

var c = new Calc();
var s = now c.twice(21);
var t = s + 1;
print("t = " ++ t);
```

```
[OK] type check passed
[method return types (inferred from reply)]
  Calc#twice -> int
[class_method_schemes]
class Calc
  twice : (int) -> int
```

パッチ前は `line 13: no overload of + matches (any, int)` で落ちていた。`now` の結果が any である限り、算術にも比較にも使えなかったためである。

A.7 s7_annotation_ok.abcl — 注釈が推論を助ける

```
// (7) 戻り値型注釈 method m(x) : T
//     注釈があれば、a も b も未束縛でも + の overload が一意に決まる。
//     s5 で principal type が無かった add が、注釈だけで解決する。
class Calc {
  method add(a, b) : int {
    reply(a + b);
  }
}

class Greeter {
  method greet(name) : string {
    reply("hello, " ++ name);
  }
  // unit を宣言したメソッドは reply しなくてよい
  method log(msg) : unit {
    print(msg);
  }
}

var calc = new Calc();
var g = new Greeter();

var s = now calc.add(1, 2);
var t = s + 1;
print("t = " ++ t);

var msg = now g.greet("AIPL");
print(msg);
```

```
[OK] type check passed
[method return types (inferred from reply)]
  Calc#add -> int
  Greeter#greet -> string
  Greeter#log -> unit
[class_method_schemes]
class Greeter
  greet : (string) -> string
  log : ('a) -> unit
class Calc
  add : (int * int) -> int
```

s5 と同じ `reply(a + b)` でありながら、曖昧警告は出ず `(int * int) -> int` に決まっている。宣言型が期待型として overload 解決へ流れた結果であり（§7.4）、引数側・戻り値側の食い違い（§15 の限界 2）も同時に解消している。

A.8 s8_annotation_conflict.abcl — 宣言と食い違う reply

```
// (8) 宣言した型と食い違う reply
//     推論のときは「他の reply とここが食い違う」としか言えなかったが、
//     注釈があると「宣言と食い違う」と言えるので blame が正しい位置に出る。
class Router {
  method route(k) : int {
    if (k > 0) { reply(1); } else { reply("negative"); }
  }
}

var r = new Router();
var v = now r.route(5);
```

```
[Type error] line 6, col 37: reply type mismatch:
  this method is declared to reply with int, but replies with string here
```

s3（注釈なし）では「他の `reply` とここが食い違う」としか言えず、どちらの分岐が誤りかを決められなかった。注釈があると「**宣言**と食い違う」と言えるので、blame が正しい位置に出る。

A.9 s9_missing_path.abcl — 全パス被覆検査

```
// (9) 全パス被覆検査 — 注釈がなければ書けない検査
//     else 側で reply しないので、k <= 0 のとき now は永久に返らない。
//     推論だけではこれを「誤り」と言えない（照合先が無いため）。
class Guard {
  method check(k) : int {
    if (k > 0) { reply(k); }
  }
}
```

```
var g = new Guard();
var v = now g.check(5);
```

```
[Type error] line 5, col 10: method check is declared to reply with int,
  but some execution path does not reply
```

`k <= 0` のとき `now g.check(...)` は永久に返らない。推論だけではこれを誤りと言えない（照合先が無い）。注釈を入れる最大の利得がこの検査である。

A.10 s10_expose_unannotated.abcl — リモート境界での注釈必須

```
// (10) リモート境界での注釈必須
//      web_expose した Adder は、本体の見えない相手から呼ばれる。
//      推論はソースが手元にある場合しか使えないので、ここは宣言でしか埋まらない。
//      → double に注釈が無いのでエラー。
//
//      Logger は web_expose していないが、web_listen があるので
//      POST /api/send?to=logger でやはり到達できる。こちらは警告。
class Adder {
  // 注釈が無いので推論に任される（-> int）。ローカルなら問題ないが…
  method double(a) {
    reply(a * 2);
  }
}

class Logger {
  method log(msg) {
    print(msg);
  }
}

var adder = new Adder();
var logger = new Logger();

web_listen(8080);
web_expose("/double", "adder");
```

```
[Type error] line 24, col 1: actor Adder is reachable from outside this
  program, but add has no return type annotation; a remote caller cannot
  see the body, so the reply type has to be declared (method add(...) : T)
```

`AIOS_LAX_EXPOSE=1` を与えるとエラーが警告に落ち、`web_expose` していない `Logger` についての警告も見える。

```
[warn] line 24, col 1: actor Adder is reachable from outside this program, ...
[warn] web_listen is called, so POST /api/send can reach any actor by name:
       actor Logger is reachable from outside this program, but log has no
       return type annotation; ...
```

```
[OK] type check passed
```

A.11　s11_service.abcl — 追加機能を一通り使う

戻り値型注釈、全パス被覆検査、双方向型付け、`select` の ρ 帰属、そして注釈を省いたメソッドの推論を 1 本にまとめたもの。

```
// (11) 今回追加した機能を全部使う統合サンプル
//
//   - method m(x) : T     戻り値型注釈
//   - 全パス被覆検査       place の if/else が両方 reply する
//   - 双方向型付け         宣言型が overload 解決へ流れる
//   - select の rho 帰属   case place(k) の reply は serve ではなく place へ返る
//   - reply からの推論     注釈の無い twice は従来どおり推論される (-> int)
//
// これを外部公開した版が s12_service_exposed.abcl。

class Stock {
  var n = 10;

  method init() {
    print("stock ready");
  }

  // n が int なので n - k の overload は一意に決まり、k も int に固定される
  method take(k) : int {
    n = n - k;
    if (n < 0) { n = 0; }
    reply(n);
  }

  method peek() : int {
    reply(n);
  }

  // reply を持たない tell 型。unit を宣言しておけば被覆検査は課されない
  method note(msg) : unit {
    print("[stock] " ++ msg);
  }
}

class Order {
  // 宣言が string なので、両方の分岐が string で reply することを検査される
  method place(k) : string {
    var left = now stock.take(k);
    if (left > 0) {
      reply("ok, left=" ++ left);
    } else {
      reply("sold out");
```

```
    }
  }

  // ★ この select の case 本体の reply は serve ではなく place への返信。
  //    eval_thread が case 実行前に set_current_msg_id を差し替えるため。
  //    したがって serve 自身は何も reply せず、unit のままでよい。
  method serve() : unit {
    print("[order] serving");
    select {
      case place(k) -> {
        var left = now stock.take(k);
        reply("served, left=" ++ left);
      }
      timeout 3000 -> {
        print("[order] idle");
      }
    }
  }

  // 注釈を省略したメソッドは従来どおり reply から推論される（-> int）
  method twice(a) {
    reply(a * 2);
  }
}

var stock = new Stock();
var order = new Order();

var s0 = now stock.peek();
print("stock = " ++ s0);

var r1 = now order.place(3);
print(r1);

var r2 = now order.place(99);
print(r2);

var t = now order.twice(21);
print("twice = " ++ t);

send stock.note("done");
```

型検査の結果。serve が unit のままであること（case 本体の reply が place に帰属していること）、注釈の無い twice が int に推論されていることに注目。

```
[OK] type check passed
[method return types (inferred from reply)]
  Order#place -> string
  Order#serve -> unit
  Order#twice -> int
```

```
  Stock#init -> unit
  Stock#note -> unit
  Stock#peek -> int
  Stock#take -> int
[class_method_schemes]
class Order
  place : (int) -> string
  serve : () -> unit
  twice : (int) -> int
class Stock
  init : () -> unit
  take : ('a) -> int
  peek : () -> int
  note : (string) -> unit
```

実際に処理系で走らせた結果（§9 の食い違いはここで見つけた）。

```
$ cat run_s11.repl
load docs/samples/reply_inference/s11_service.abcl
compile
$ ./src/abclrepl_thread -q -f run_s11.repl
[Defined class Stock]
[Registered types for class Stock: init/0, take/1, peek/0, note/1]
[Defined class Order]
[Registered types for class Order: place/1, serve/0, twice/1]
stock ready
stock = 10
ok, left=7
sold out
twice = 42
[stock] done
```

A.12　s12_service_exposed.abcl — 同じものを外部公開する

s11 に `web_listen` と `web_expose` を足しただけ。ローカルなら推論で済んでいた `twice` が、公開した途端に宣言を要求される。

```
web_listen(8080);
web_expose("/order", "order");
web_expose("/stock", "stock");
```

```
[Type error] line 84, col 1: actor Order is reachable from outside this
  program, but twice has no return type annotation; a remote caller cannot
  see the body, so the reply type has to be declared (method twice(...) : T)
```

`twice` に `: int` を付けるだけで通る。これが §3 で述べた「推論はソースが手元にある場合しかカバーできない」ということの具体形である。

A.13　s13_runtime_mismatch.abcl — 型検査器と評価器の食い違い

```
// (13) 型検査器と評価器の食い違い ―― 発見と修正の回帰テスト
//
//      発見時:
//        型検査  [OK]  T#f -> int
//        実行時  3.          ← VFloat。|- e : int なのに e ⊠ VFloat で
//                               preservation が破れていた。
//
//      原因は eval_thread の apply_binop が + - * / を整数どうしでも
//      VFloat で返していたこと。typing_env は + : (int * int) -> int を
//      持っていたので、両者が食い違っていた。
//
//      修正 (a):
//        - eval_thread : VInt 同士の + - * は VInt を返す
//        - typing_env  : / だけ (int * int) -> float に型付けし直す
//                        (整数除算は行わない。7 / 2 は 3.5 のまま)
//
//      現在:
//        型検査  [OK]  T#f -> int
//        実行時  3
//
//      なお a + b の両辺が未束縛の型変数のときは principal type が無く、
//      型検査は float を選んで警告を出す一方、実行時に int が渡れば
//      VInt が返る。この食い違いだけは残っているが、対象は
//      「ambiguous overload」と警告されるプログラムに限られ、
//      戻り値型を注釈すれば解消する (s7 を参照)。
class T {
  method f() : int {
    reply(1 + 2);
  }
}

var t = new T();
var v = now t.f();
print(v);
```

```
$ ./tc docs/samples/reply_inference/s13_runtime_mismatch.abcl
[OK] type check passed
[method return types (inferred from reply)]
  T#f -> int

$ ./src/abclrepl_thread -q -f run_s13.repl
3
```

修正前はここが 3. だった（§9）。

A.14　s14_select_pattern.abcl — select パターンの照合

```
// (14) select パターンをメソッド署名に照合する
//
//      case m(x) は method m(a, b) 宛のメッセージを受けるつもりで書かれているが、
//      束縛する変数が1つしかない。実行時には match_callstmt が
//      arity 不一致で「一致しない」と判断するので、この case は永久に選ばれず、
//      timeout に落ち続ける。静かに壊れる典型。
//
//      これまではパターン変数に fresh な型変数を振るだけで署名を見ていなかった
//      ため素通りしていた。いまは arity と引数型を照合する。
class C {
  method m(a, b) : int {
    reply(a + b);
  }

  method loop() : unit {
    select {
      case m(x) -> { reply(x); }
      timeout 100 -> { print("timeout"); }
    }
  }
}

var c = new C();
```

```
[Type error] line 17, col 22: select: case m binds 1 variable(s)
  but method m takes 2
```

A.15　s15_double_reply.abcl — reply の線形性

```
// (15) reply の線形性 —— 二重 reply を弾く
//
//      resolve_future は 2 度目の呼び出しでも黙って fresult を上書きするだけで、
//      待ち手はすでに 1 度目の値を受け取ったあとなので、2 度目の値は捨てられる。
//      エラーにもならないので、書いた側は気づかない。
//
//      注釈の有無によらず「高々一度」を要求する。
//      注釈があるメソッドは「全パスで少なくとも一度」も課されるので、
//      合わせて「ちょうど一度」になる。
class C {
  method m(k) : int {
    reply(k);
    reply(k + 1);
  }
}
```

```
var c = new C();
var r = now c.m(1);
print(r);
```

```
[Type error] line 11, col 10: method m may reply more than once on some path;
  a method must reply at most once
```

ループ本体の `reply` も同じく拒否される。

```
method m(k) : int { while k > 0 do { reply(k); } }
```

```
[Type error] method m may reply more than once on some path; ...
```

一方、分岐ごとに一度ずつ返す正しい形は通る。

```
method m(k) : int { if (k > 0) { reply(1); } else { reply(2); } }
```

A.16　s16_concat.abcl — 文字列連結 ++ と数値 +

```
// (16) 文字列連結 ++ と数値 +
//
//      かつて + は数値2種と文字列連結2種の計4候補を持っていた。
//      両辺が未束縛だと4つとも一致してしまい、しかも候補列が登録順の逆
//      だったため ('a * string) -> string が必ず先頭に来て、
//      a + b が問答無用で string に潰れていた。
//
//      文字列連結を ++ に分けたので、
//        + は数値専用（候補2つ、int と float）
//        ++ は両辺を文字列化して連結（候補1つ、曖昧さなし）
//      になった。++ の型は forall a b. (a * b) -> string である。
class C {
  // ++ は左右どちらが文字列でなくてもよい
  method label(n) : string {
    reply("n=" ++ n);
  }
  method both() : string {
    reply(1 ++ 2);          // "12"
  }
  method plain() : string {
    reply("a" ++ "b");      // "ab"
  }
  // + は数値のまま。文字列を渡せば型エラーになる
  method sum() : int {
    reply(1 + 2);           // 3
  }
}

var c = new C();
print(now c.label(42));
```

```
print(now c.both());
print(now c.plain());
print(now c.sum());
```

```
[OK] type check passed
[method return types (inferred from reply)]
  C#both  -> string
  C#label -> string
  C#plain -> string
  C#sum   -> int

$ ./src/abclrepl_thread -q -f run_s16.repl
n=42
12
ab
3
```

付録 B　型検査ドライバ tc_main.ml

REPL（SDL に依存する）を起動せずに型検査だけを走らせるための最小ドライバ。

```
(* tc_main.ml — AIPL/ABCL の型検査だけを走らせる小さなドライバ *)

let parse_file (fname : string) : Ast.program =
  let ic = open_in fname in
  let len = in_channel_length ic in
  let s = really_input_string ic len in
  close_in ic;
  let lb = Lexing.from_string s in
  lb.Lexing.lex_curr_p <- { lb.Lexing.lex_curr_p with Lexing.pos_fname = fname };
  Parser.program Lexer.token lb

let () =
  let fname = Sys.argv.(1) in
  Printf.printf "=== %s ===\n%!" (Filename.basename fname);
  match (try Ok (parse_file fname) with e -> Error (Printexc.to_string e)) with
  | Error m -> Printf.printf "[Parse error] %s\n%!" m; exit 2
  | Ok prog ->
      (* クラスメソッドのスキーム表は出さず、戻り値型の表だけ見たいので
         Infer.verbose は false にして、自前で表示する *)
      Infer.verbose := false;
      (match Infer.check_program prog with
       | Ok _ ->
           print_endline "[OK] type check passed";
           Types.debug_print_method_rets ();
           print_endline "[class_method_schemes]";
           Hashtbl.iter
             (fun cls sigs ->
                Printf.printf "class %s\n" cls;
```

```
            List.iter
              (fun (m, sch) ->
                 Printf.printf "  %s : %s\n" m
                   (Types.string_of_ty_pretty (Types.repr (Types.instantiate sch))))
              sigs)
         Types.class_method_schemes
   | Error msg ->
       Printf.printf "[Type error] %s\n" msg;
       Types.debug_print_method_rets ());
  print_newline ()
```

付録 C　変更ファイル

- src/types.ml（+41）— ρ 表（method_ret_tys と register / lookup / reset）、debug_print_method_rets。
- src/infer.ml（+409 / -67）— ρ の確保と伝播、check_reply、全パス被覆検査 replies_on_all_paths、期待型の伝播（expected_of と infer_expr ?expected）、pick_overload の修正、リモート境界の検査 check_boundary_annotations、select の case における ρ の帰属修正。
- src/ast.ml（+13 / -3）— method_decl に ret : Types.ty option を追加。
- src/parser.mly（+26 / -1）— 注釈つき method_decl 規則と ret_ann、型名を写す ty_of_name。
- src/lexer.mll（+1）— ":" → COLON。
- src/Makefile（1 行）— リンク順を修正（types.cmo を ast.cmo より前へ）。
- src/repl_thread.ml（+1）— トークン表示に COLON を追加。
- src/eval_thread.ml（+14）— VInt どうしの + - * を VInt で返す（§9）。
- src/typing_env.ml（+4 / -1）— / を (int × int) → float に型付け。
- scripts/type_runtime_diff.py（新規）— 型と実行値の差分テストハーネス（§11.3）。
- 文字列連結の ++ への分離（§12）— lexer.mll、parser.mly、typing_env.ml、infer.ml、eval_thread.ml。
- abclc/*.abcl ほか 38 ファイル / 255 箇所 — + を ++ へ移行。
- src/infer.ml — 曖昧 overload を既定でエラーに、仮引数を署名へ結線（§13）。
- abclc/Hello.abcl — new Hello(5) → new Hello(5.)（結線で露見した int/float 不整合）。
- abclc/web_calc.abcl, src/web_calc.abcl, src/web_calc1.abcl — 公開アクターに戻り値型注釈を付けて移行。

付録 D　参照

- Pony Tutorial, *Methods*. https://tutorial.ponylang.io/expressions/methods.html
- ponylang/rfcs #0010, *Generic Type Inference*. https://github.com/ponylang/rfcs/blob/main/text/0010-generic-type-inference.md

- Fernandez-Reyes et al., *An Optimised Flow for Futures: From Theory to Practice*. `https://arxiv.org/pdf/2107.07298`
- de Boer et al., *Forward to a Promising Future*. `https://link.springer.com/chapter/10.1007/978-3-319-92408-3_7`
- 本リポジトリ `docs/ABCLCP_TYPE_SOUNDNESS_REPORT.md`（§5, §12）