ABCL/c+ (AIPL) の型システムと健全性証明

実装に基づくレポートと Coq 形式化

Yasushi Kodama Repository Report

2026 年 5 月 4 日

概要

本レポートは、GitHub リポジトリ `https://github.com/yaskodama/abclcp.git` の現在実装に基づき、ABCL/c+、すなわち AIPL 的に拡張されつつある actor-first 言語の型システムを整理し、その健全性を Coq で検証可能なコア体系として示す。対象ブランチは `make-src-base`、参照コミットは `fa93ee0 Save current version` である。

現在の実装は Hindley-Milner 風の型変数、型スキーム、単一化、primitive overload、actor method registry、`now`、`future`/`await`、および runtime session protocol checker を備える。一方、`any`、remote actor、`send!`、`sender`、method return type、静的 session type には未完成の動的境界が残る。したがって本レポートでは、現在実装が保証している静的安全性と、runtime protocol checker が保証する線形順序安全性を分けて定式化する。

Coq 形式化は `coq/AbclSoundness.v` にあり、The Rocq Prover version 9.1.0 の `coqc` compatibility binary により検査済みである。

目次

1　対象と方法

本レポートの対象は、ローカル作業ツリー/Users/kodamay/aios/abclcp にある yaskodama/abclcp である。調査時点での remote は次であった。

```
origin https://github.com/yaskodama/abclcp.git
```

対象ブランチは make-src-base であり、origin/make-src-base と同期済みであった。未追跡ファイルとして docs/RETURN_PROMPT.md が存在したが、本作業では変更していない。

調査した主要ファイルは表 1 の通りである。

表 1: 主要実装ファイル

ファイル	役割
src/ast.ml	式、文、class、method、send target、select case の AST 定義
src/parser.mly	ABCL/c+ の具象構文
src/lexer.mll	lexer 定義
src/types.ml	型、型スキーム、単一化、class method registry
src/typing_env.ml	primitive と overload を持つ初期型環境
src/infer.ml	型推論と型検査
src/eval_thread.ml	actor runtime、mailbox、future、reply、session protocol runtime
abclc/*.abcl	サンプルプログラム
docs/AIOS_LANGUAGE_DESIGN.md	AIOS 言語化に向けた設計文書

2　言語の実行モデル

ABCL/c+ は actor を第一級の実行単位とする。現在の README では、AIOS 上の kernel service、UI component、web endpoint、device adapter、model adapter、memory service、tool adapter、autonomous agent をすべて actor として扱う方向性が示されている。

実行時の actor は概念的に以下を持つ。

- actor name
- class または behavior name
- mailbox
- local environment
- method table
- last sender
- message id と session id を含む実行コンテキスト

message は actor から actor へ送られる。通常の `send` は fire-and-forget であり、`future` は message id と future object を対応づけ、`reply` によって結果を resolve する。`now` は runtime 上は `future` を作って即 `await` する同期 send として実装されている。

3 抽象構文

`src/ast.ml` に基づく式の中核は次である。

$$
\begin{array}{rcl}
e & ::= & n \mid f \mid s \mid x \mid e_1\ op\ e_2 \\
& \mid & f(e_1, \ldots, e_n) \mid \texttt{new}\ C(e_1, \ldots, e_n) \\
& \mid & [e_1, \ldots, e_n] \mid \texttt{now}\ t.m(e_1, \ldots, e_n) \\
& \mid & \texttt{future}\ t.m(e_1, \ldots, e_n) \mid \texttt{await}\ e
\end{array}
$$

文の中核は次である。

$$
\begin{array}{rcl}
s & ::= & x = e \mid \texttt{var}\ x = e \mid f(e_1, \ldots, e_n) \\
& \mid & \texttt{send}\ t.m(e_1, \ldots, e_n) \mid \texttt{send!}\ t.m(e_1, \ldots, e_n) \\
& \mid & \texttt{become}\ C(e_1, \ldots, e_n) \mid s_1; s_2 \\
& \mid & \texttt{if}\ e\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ e\ \texttt{do}\ s \\
& \mid & \texttt{select}\ \{case_i, timeout?\}
\end{array}
$$

send target はローカル actor 名、または `remote("host:port", "actor")` である。`self` と `sender` は parser と型検査器で特別扱いされる。

4 型の構文

`src/types.ml` の型定義は、以下の型構文として読める。

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

`TVar` は mutable link を持ち、単一化によって別の型へ束縛される。`repr` は代表型を辿り、経路圧縮を行う。単一化 `unify` は基底型の一致、関数型の arity と引数・戻り値の一致、配列要素型、future 型を検査し、occurs check によって無限型を拒否する。

5 型スキームと型環境

型スキームは

$$\sigma ::= \forall \alpha_1, \ldots, \alpha_n.\tau$$

であり、実装では `Forall of int list * ty` である。`generalize` は型環境に自由でない型変数を量化し、`instantiate` は量化変数を fresh な型変数へ置き換える。

型環境は名前から型スキームのリストへの写像である。

$$\Gamma : name \to list(\sigma)$$

リストになっているのは overload のためである。例えば `+` は `int * int -> int`、`float * float -> float`、`string * 'a -> string`、`'a * string -> string` を持つ。`pick_overload` は引数型と単一化可能な候補を探し、戻り値型を返す。

6 Actor 型

actor 型は class 名と method signature の表を持つ。

$$\texttt{actor}(C)\{m_1 : \tau_1; \ldots; m_n : \tau_n\}$$

現在の実装では、program 全体を検査する前に `preinfer_all_classes` が class 群を走査し、各 method の型スキームを `class_method_schemes` に登録する。method 型は現状では引数に fresh 型変数を割り当て、戻り値を `unit` として登録する。

$$m(x_1, \ldots, x_n) : (\alpha_1 * \cdots * \alpha_n) \to \texttt{unit}$$

runtime では method 本体が `reply(v)` によって値を返すことができる。しかし AST には method return annotation がなく、型検査器も `reply` の値型を method 戻り値型へ接続していない。したがって、現在の `now` と `future` の戻り値は静的には `any` として近似される。

7 主要な型規則

7.1 変数

$$\frac{x : \sigma \in \Gamma \quad instantiate(\sigma) = \tau}{\Gamma \vdash x : \tau}$$

本番の型検査では未束縛変数はエラーである。ただし preinfer phase では、未束縛変数に fresh 型変数を与えて class method registry の構築を先に進める。

7.2 primitive call と overload

$$\frac{\Gamma \vdash e_i : \tau_i \quad pick\_overload(f, \tau_1, \ldots, \tau_n) = \tau}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

演算子も primitive 関数として扱われる。ただし文字列連結では、`+` の片側が `string` なら戻り値を `string` とする特別処理がある。

7.3 配列

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_i : \tau \ (2 \le i \le n)}{\Gamma \vdash [e_1, \ldots, e_n] : \tau[]}$$

空配列は現在 `unit[]` として扱われる。

7.4 new

$$\frac{C.init : (\tau_1 * \cdots * \tau_n) \to \texttt{unit} \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n) : \texttt{actor}(C)}$$

`init` が存在する場合、arity と引数型を単一化する。`init` が存在しない場合、現在の実装では constructor 引数の検査は省略される。

7.5 send

ローカル actor への通常 send は次を検査する。

$$\frac{\Gamma \vdash target : \texttt{actor}(C) \quad C.m : (\tau_1 * \cdots * \tau_n) \to \tau_r \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{send}\ target.m(e_1, \ldots, e_n)\ ok}$$

現在の実装では method return type は send 文に使われない。検査対象は target が actor であること、method が存在すること、arity と引数型が合うことである。

7.6 send!

`send!` は unsafe send である。

$$\frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{send!}\ target.m(e_1, \ldots, e_n)\ ok}$$

target の actor 型、method の存在、payload shape は検査されない。このため `send!` は型健全性定理の静的フラグメントから除外する。

7.7 future と await

設計上望ましい型規則は次である。

$$\frac{\Gamma \vdash target : \texttt{actor}(C) \quad C.m : (\tau_1 * \cdots * \tau_n) \to \tau_r \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{future}\ target.m(e_1, \ldots, e_n) : \texttt{future}\ \tau_r}$$

$$\frac{\Gamma \vdash e : \texttt{future}\ \tau}{\Gamma \vdash \texttt{await}\ e : \tau}$$

現在の実装では local target の method 存在と引数型を検査したうえで、戻り値型は `future any` になる。remote future send は runtime で未実装であり、future を rejected にする。

7.8 now

設計上の `now` は同期 send であり、method return type を返す。

$$\frac{\Gamma \vdash target : \texttt{actor}(C) \quad C.m : (\tau_1 * \cdots * \tau_n) \to \tau_r \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{now}\ target.m(e_1, \ldots, e_n) : \tau_r}$$

現在の実装では `now` は `future` を作って即 `await` するため、型は `any` として扱われる。

7.9 `become`

$$\frac{C.init : (\tau_1 * \cdots * \tau_n) \to \texttt{unit} \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{become}\ C(e_1, \ldots, e_n)\ ok}$$

現在は behavior 切り替え前後の protocol compatibility と、pending mailbox に残っている message との整合性は検査されない。

7.10 `select`

$$\frac{\Gamma, x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash body\ ok}{\Gamma \vdash case\ m(x_1, \ldots, x_n) \to body\ ok}$$

各 case の束縛変数は fresh 型変数として body 内に導入される。現在の `select` は mailbox から method 名に合う message を選ぶ runtime 構文であり、session type とはまだ静的に接続されていない。

8 セッション型と現在の protocol runtime

現在実装における「セッション型」は、厳密には静的型ではなく、runtime の線形 protocol checker である。`src/eval_thread.ml` では protocol を actor.method の列として定義する。

$$P ::= a_1.m_1 \to a_2.m_2 \to \cdots \to a_n.m_n$$

例として `abclc/session_protocol_solve.abcl` は次の protocol を使う。

```
protocol_define("SolveProtocol",
  "planner.plan -> solver.solve -> reviewer.review");
var sid = protocol_start("SolveProtocol");

var plan = now planner.plan("3 boxes with 4 apples");
var answer_future = future solver.solve(plan);
var answer = await answer_future;
var verdict = now reviewer.review(answer);
protocol_end(sid);
```

runtime session state は session id、protocol definition、現在位置 `pos`、closed flag を持つ。send の直前に `protocol_check_send` が呼ばれ、現在位置の expected step と送信先 actor/method が一致すれば `pos` を 1 進める。一致しなければ例外を投げ、message は mailbox に入らない。

この runtime checker が保証する性質は以下である。

1. 順序保存: checked send は protocol 定義の順にしか進まない。
2. 線形消費: 1 回の checked send は position をちょうど 1 進める。
3. 完了後拒否: 全 step 消費後の追加 send は拒否される。
4. 不完全終了拒否: 未消費 step がある状態で `protocol_end` すると失敗する。
5. mailbox 汚染防止: 違反 message は actor mailbox に入る前に拒否される。

9 Coq 形式化

Coq 形式化は `coq/AbclSoundness.v` にある。検査コマンドは次である。

```
opam exec -- coqc coq/AbclSoundness.v
```

実行環境では次を確認した。

```
The Rocq Prover, version 9.1.0
compiled with OCaml 5.1.1
```

形式化は実装全体ではなく、安全性の核を抽出した小さな体系である。対象は first-order value、`future`/`await`、および線形 session protocol checker である。

9.1 式コア

Coq 側の型は次である。

```
Inductive ty : Type :=
| TNat
| TString
| TUnit
| TAny
| TFuture (t : ty).
```

式は次である。

```
Inductive expr : Type :=
| ENat (n : nat)
| EString (s : string)
| EUnit
| EAdd (e1 e2 : expr)
| EFutureValue (v : expr)
| EAwait (e : expr).
```

この閉じた式体系に対して、標準的な progress と preservation を証明した。

定理 1 (Preservation)**.** もし $\vdash e : \tau$ かつ $e \to e'$ なら、$\vdash e' : \tau$ である。

定理 2 (Progress)**.** もし $\vdash e : \tau$ なら、e は値であるか、ある e' が存在して $e \to e'$ である。

9.2 セッション protocol checker

Coq 側では label を actor 名と method 名の組として定義する。

```
Record label : Type := {
  actor_name : string;
  method_name : string
}.
```

protocol は label の list であり、`send_ok p pos l pos'` は `nth_error p pos = Some l` かつ `pos' = S pos` のときだけ成り立つ。

```
Inductive send_ok : protocol -> nat -> label -> nat -> Prop :=
| SendOk : forall p pos l,
    nth_error p pos = Some l ->
    send_ok p pos l (S pos).
```

証明済み定理は次である。

- `send_ok_unique`: 同一 protocol、position、label に対する次 position は一意。
- `send_ok_advances_one`: checked send は position をちょうど 1 進める。
- `complete_rejects_next`: 完了位置では追加 send は存在しない。
- `wrong_label_rejected`: expected label と異なる label は受理されない。
- `send_ok_within_protocol`: accepted send は protocol 範囲内でのみ発生する。

10 健全性主張

現在の ABCL/c+ に対して、完全な “well-typed programs cannot go wrong” をそのまま主張するのは正確でない。理由は `any`、remote actor、`send!`、外部 primitive、AI provider、thread scheduler、mutex、network が存在するためである。

正確な主張は次である。

> well-typed programs do not get stuck inside the statically typed fragment; dynamic boundaries may fail, but failures are explicit at the boundary.

静的フラグメントに含まれるものは、基底型、配列、primitive overload、local actor send の target/method/arity/payload 検査、`await` の future 型検査である。動的境界に含まれるものは、`any`、`sender`、remote actor、`send!`、method return type と `reply` の未接続部分である。

11 静的セッション型への拡張

現在の runtime protocol を静的 session type へ発展させるには、protocol string を AST と型環境へ昇格する必要がある。例えば次のような構文を導入する。

```
protocol SolveProtocol {
  planner.plan(string) -> string;
  solver.solve(string) -> string;
  reviewer.review(string) -> string;
}
```

このとき型判断は通常の型環境 Γ に加え、session environment Δ を持つ。

$$\Gamma; \Delta \vdash s \triangleright \Delta'$$

次 action が $a.m$ の場合、send は protocol tail へ session environment を進める。

$$\frac{\Delta(sid) = a.m : (\tau_1 * \cdots * \tau_n) \to \tau_r; P \quad \Gamma \vdash e_i : \tau_i}{\Gamma; \Delta \vdash \texttt{now}\ a.m(e_1, \ldots, e_n) : \tau_r \triangleright \Delta[sid := P]}$$

この設計により、以下の定理を目標にできる。

定理 3 (Session Preservation)**.** *well-typed send* は *session environment* を *protocol tail* へ進める。

定理 4 (Protocol Fidelity)**.** *well-typed closed program* の *runtime trace* は、宣言された *protocol* の *prefix* である。

12 実装上の次の課題

現在の型システムをより強くするための最重要課題は、method return type を AST と型検査器へ導入することである。これにより `reply`、`now`、`future`、`await` を単一の戻り値型で結べる。

1. method 構文に return annotation を追加する。
2. `reply(v)` の型を current method return type と照合する。
3. `future target.m(args)` を `future` τ_r にする。
4. `now target.m(args)` を τ_r にする。
5. protocol を string ではなく AST として parse する。
6. session environment を型判断へ導入する。
7. `any` 境界を gradual typing または contract として整理する。

13 結論

ABCL/c+ の現在の型システムは、AIOS actor 言語として意味のある基盤を持つ。基底型、多相 primitive、array、future、actor method registry、local send 検査により、message passing の多くを静的に検査できる。runtime session protocol はまだ静的型ではないが、線形 protocol checker として設計されており、Coq で示した単純な automaton 不変条件に支えられている。

今後 method return type と静的 session environment を導入すれば、ABCL/c+ は actor の

局所的型安全性と workflow の大域的順序安全性を同じ型システムで扱える AIOS 言語へ発展できる。

付録 A　検査コマンド

Coq 証明の検査:

```
opam exec -- coqc coq/AbclSoundness.v
```

本 LaTeX レポートの DVI 生成:

```
cd paper
uplatex abclcp_type_soundness.tex
uplatex abclcp_type_soundness.tex
```

DVI から PDF への確認用変換:

```
dvipdfmx abclcp_type_soundness.dvi
```