関数型まつり 2026

2026 年に読む

The Definition of Standard ML

現代の堅牢なソフトウェア設計の源流として

芳賀 雅樹 / @silasolla

自己紹介

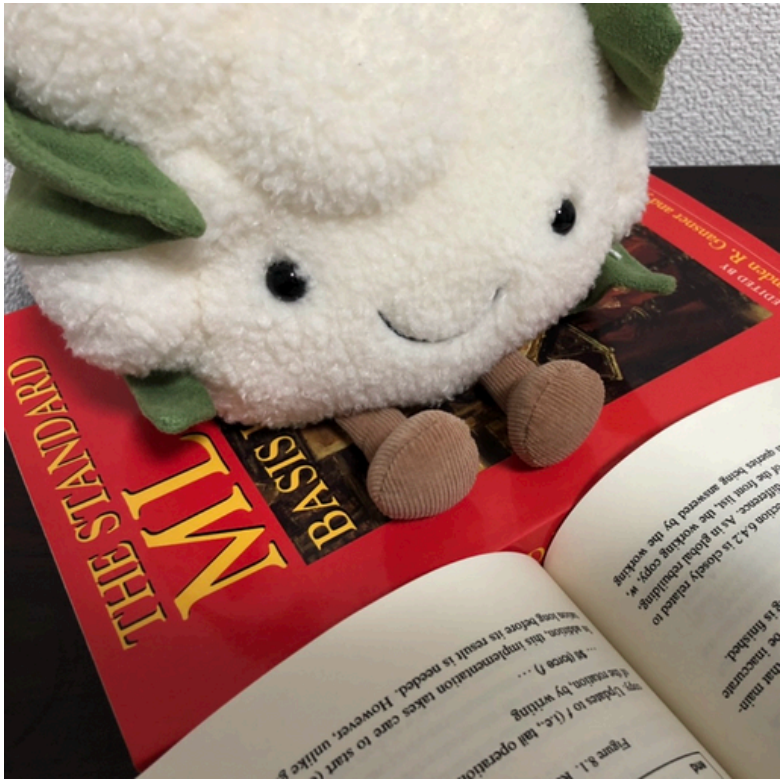
芳賀 雅樹 / @silasolla

Standard ML がすきです！

- 当然 Machine Learning ではなく Meta Language……！

3-shake Inc. で働いています！

- アプリ開発のモダナイゼーション支援をやっている
- トイルを無くすのが生業



インターネット近影

はじめに

数値と文字列を足してみる

```
// JavaScript
const x = 1 + "hello";  // "1hello"（暗黙の型変換）

# Python
x = 1 + "hello"  # 実行時に TypeError

// Rust
let x = 1 + "hello";  // コンパイルエラー
```

なんで型変換される？

`1 + "hello"` での暗黙の型変換は JavaScript の「仕様」

> If lprim is a String or rprim is a String, then
>
> i. Let lstr be ? ToString(lprim).
> ii. Let rstr be ? ToString(rprim).
> iii. Return the string-concatenation of lstr and rstr.
>
> どちらかが String なら，両方を String にして連結する
>
> —— ECMA-262 §13.15.3

プログラミング言語の定義

処理系が事実上の仕様

- Python (CPython)
- Ruby (MRI)
- Rust (rustc)

動くものの振る舞いが仕様 (マニュアルは補助)

自然言語の仕様書

- JavaScript (ECMA-262)
- Java (JLS)
- C (ISO)

散文の要求を解釈して各々が実装

形式的な推論規則

SML (The Definition)

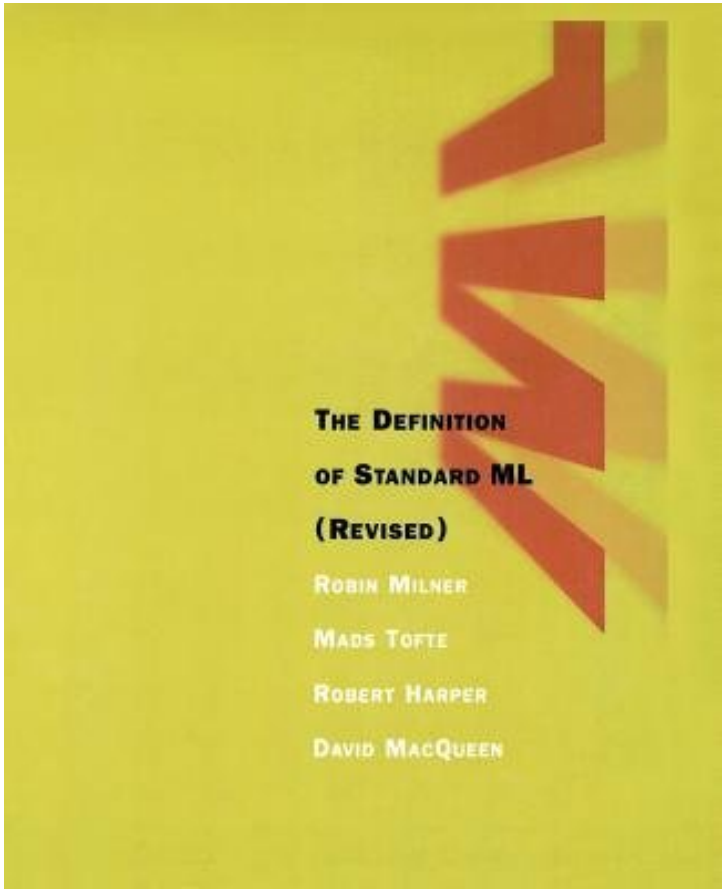
自然意味論で記述 (形式的に導出される)

The Definition of Standard ML

言語の振る舞いを **推論規則の集まり** として形式化

こういう規則で定義される：

$$\frac{C \vdash \mathrm{dec} \Rightarrow E \quad C \oplus E \vdash \mathrm{exp} \Rightarrow \tau}{C \vdash \mathrm{let\ dec\ in\ exp\ end} \Rightarrow \tau}$$



The Definition of Standard ML (Revised), 1997

なんでこんなことする？

コーディングするだけなら，散文と BNF のリファレンスマニュアルで事足りる

プログラミング言語そのものを扱う人まで想定し，文法だけでなく形式的な意味も記述した

> … a robust program written in an insecure language
> is like a house built upon sand.
>
> 堅牢なプログラムも安全でない言語で書かれていたら，砂上の楼閣のようなもの……
>
> —— Preface

定義であって実装ではない

- 型推論アルゴリズム (Algorithm W, J)
  —— 含まれていない
- 構文解析アルゴリズム
  —— 含まれていない
- メモリ管理 (ヒープ ／ スタック ／ GC)
  —— 含まれていない

仕様はあるが，実装は自由

Nano Banana 2 による一句

これから見ていくこと

✗ SML を書くための文法や機能を紹介！

✓ 形式的な仕様が **何を定めたか ／ 定めなかったか**

意味論の分離

型情報が消える言語

```typescript
// TypeScript
const add = (a: number, b: number): number => a + b;
```

⬇

```javascript
// JavaScript（コンパイルで型が消える）
const add = (a, b) => a + b;
```

実行時に型情報はいらない？

動的型付き言語 (JS / Python)

値の計算で型情報 (動的型タグ) を見る

```
// 実行時に型の暗黙変換
const x = 1 + "hello"
// => "1hello"

# 実行時に型で弾かれる
x = 1 + "hello" # => TypeError
```

静的型付き言語 (SML)

値の計算に型情報は「不要」← ？

```
(* コンパイル時に型エラー *)
val x = 1 + "hello"
(* Error: operator and operand
   don't agree *)
```

Standard ML の意味論

静的意味論 (Elaboration)

プログラムの「正当性」を判定

$C \vdash \mathrm{exp} \Rightarrow \tau$

文脈 C で式 exp は型 τ に精緻化

動的意味論 (Evaluation)

プログラムの「計算結果」を定義

$E \vdash \mathrm{exp} \Rightarrow v$

環境 E で式 exp は値 v に評価

どちらも $A \vdash \mathrm{phrase} \Rightarrow A'$ という形 (A' は環境の場合も)

推論規則の読み方

上部に **仮定** ／ 下部に **結論** を記述：

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_n}{C}$$

仮定 $P_1 \cdots P_n$ が全て成立すれば，結論 C も成立する

静的意味論の規則

let 式 —— Rule (4), p.25

$$\frac{C \vdash \mathrm{dec} \Rightarrow E \quad C \oplus E \vdash \mathrm{exp} \Rightarrow \tau}{C \vdash \mathbf{let}\,\mathrm{dec}\,\mathbf{in}\,\mathrm{exp}\,\mathbf{end} \Rightarrow \tau}$$

関数適用 —— Rule (8), p.25

$$\frac{C \vdash \mathrm{exp} \Rightarrow \tau' \to \tau \;\; C \vdash \mathrm{atexp} \Rightarrow \tau'}{C \vdash \mathrm{exp}\,\mathrm{atexp} \Rightarrow \tau}$$

型 τ が付く条件を定義する (C や E の中にも型情報が含まれる)

動的意味論の規則

let 式 —— Rule (93), p.49

$$\frac{E \vdash \mathrm{dec} \Rightarrow E' \quad E + E' \vdash \mathrm{exp} \Rightarrow v}{E \vdash \mathbf{let}\,\mathrm{dec}\,\mathbf{in}\,\mathrm{exp}\,\mathbf{end} \Rightarrow v}$$

関数適用 —— Rule (102), p.49

$$\frac{E \vdash \mathrm{exp} \Rightarrow (\mathrm{match}, E', \mathrm{VE})\ E \vdash \mathrm{atexp} \Rightarrow v\ E' + \mathrm{Rec\ VE}, v \vdash \mathrm{match} \Rightarrow v'}{E \vdash \mathrm{exp\,atexp} \Rightarrow v'}$$

値 v, v' に計算される条件を定義する (型情報は VE や E にも含まれず)

値の計算に型は不要

$$\frac{E \vdash \mathrm{exp} \Rightarrow v_1 \quad \mathrm{is\_function}(v_1) = \mathrm{false}}{E \vdash \mathrm{exp}\,\mathrm{atexp} \Rightarrow \mathrm{TypeError}}$$

こんな規則は存在しない！

Note that none of the rules for function application has a premise in which the operator evaluates to a constructed value, a record or an address.

関数適用の推論規則に，関数でない値が適用されるケースはない

—— p.49

安全性の所在は？

評価の規則は型を見ないので，実行時に型情報は **いらない**

「精緻化 (型検査)」と「評価」を形式化したということは……

「型検査に通れば実行時の評価は破綻しない」ことを書ける！ (**型安全性**)

証明は後続の研究：

Milner & Tofte (1991) ～ Lee, Crary & Harper (2007) の機械検証

型の抽象化

何にでも文字列を使うと？

ユーザ ID もメールアドレスも string だと……

```
function updateEmail(userId: string, email: string): void { ... }

updateEmail(email, userId);   // 逆に渡してもコンパイルが通る
```

Branded Types

(構造的型付けなので) こういうテクニックやりがち

```
type UserId = string & {
  readonly __brand: unique symbol
};

const uid = "ユーザじゃないよん" as UserId;    // 簡単に剥がれる
```

言語ではなく，レビューや lint や慣習など (書き手の技術が試される……)

公称型があれば充分？

宣言ごとに新しい型名を導入

```rust
// Rust
struct UserId(String); // newtype
struct Email(String);
```

```sml
(* SML *)
datatype user_id = UserId of string
datatype email   = Email  of string
```

どこでも UserId を触れてしまったら？

- UserId "うにょーん"
- UserId ""

型を隠蔽する

structure (実装の塊)

```
type t = string
val prefix = "u_"
fun create s = prefix ^ s
```

:>

signature (インターフェース)

```
type t ―― 抽象型 (string を隠蔽)
val create : string -> t
```

シグネチャにない束縛 (**prefix**) は，外からは存在しなくなる

type t とだけ書くと **t = string** だった事実ごと **型環境から消える**

不透明シグネチャ

```
signature USER_ID = sig
  type t
  val create : string -> t
end

structure UserId :> USER_ID = struct
  type t = string
  val prefix = "u_"
  fun create s = prefix ^ s
end

val x = UserId.create "hello"      (* OK: create 経由 *)
val y : UserId.t = "hello"         (* Error: t ≠ string *)
```

推論規則を見てみる

不透明シグネチャ —— Rule (53), p.39

$$\frac{B \vdash \text{strexp} \Rightarrow E \quad B \vdash \text{sigexp} \Rightarrow (T')E'}{B \vdash \text{strexp} :> \text{sigexp} \Rightarrow E'}$$

条件：$(T')E' \geq E'' \prec E$, $T' \cap (T \text{ of } B) = \varnothing$

新しい型名 T' は，現在の Basis B に **存在しない名前** でなければならない

推論規則に UserId を通してみる

仮定 1 $B \vdash \mathrm{strexp} \Rightarrow E$　ストラクチャの精緻化 (t の実体が見える)

```
struct
  type t = string
  val prefix = "u_"
  fun create s = prefix ^ s
end
```

$\longrightarrow$

$E =$ { t = string, prefix : string,
create : string -> string }

仮定 2 $B \vdash \mathrm{sigexp} \Rightarrow (T')E'$　シグネチャの精緻化 (新しい型名 t' を生成)

```
sig
  type t
  val create : string -> t
end
```

$\longrightarrow$

$T' =$ { t' }
$E' =$ { t = t', create : string -> t' }

推論規則に UserId を通してみる (つづき)

条件 $(\{t'\})E' \geq E'' \prec E \quad \varphi = \{\ t' \mapsto \texttt{string}\ \}$ を選んで E'' (インスタンス) を構成

$$E'' = \varphi(E') = \{\ \texttt{t = string, create : string -> string}\ \}$$

$E'' \prec E$ は E'' の各成分が E に含まれることを要求 (余分な `prefix` は問題なし)

結論 $B \vdash \mathrm{strexp} :> \mathrm{sigexp} \Rightarrow E' \quad \varphi$ を通す前の E' が付く

$$E' = \{\ \texttt{t = } t' \texttt{, create : string -> } t'\ \}$$

型の隠蔽は結論が E' であるという **推論規則の形** そのもの

補足：シグネチャの種類

互換性が価値になる場合は透明シグネチャを使う場合もある

```
structure Impl = struct
  type t = string
  val prefix = "u_"
  fun create s = prefix ^ s
end

structure Trans : USER_ID = Impl    (* 透明 *)
val a : Trans.t = "hello"           (* OK: Trans.t = string *)

structure Opaque :> USER_ID = Impl (* 不透明 *)
val b : Opaque.t = "hello"          (* Error: Opaque.t ≠ string *)
```

型の依存解決

DI コンテナ

```
container.bind<Logger>("Logger").to(ConsoleLogger);
const logger = container.get<Logger>("Logger");
```

```
Error: Cannot resolve dependency <Logger>
  required by UserService
```

典型的には **起動時** に，遅延注入なら該当コードの **実行時** に落ちる

ファンクタ (Haskell とは別物)

ストラクチャを受け取って，新しいストラクチャを返す関数

```
signature LOGGER = sig
  val log : string -> unit
end

(* 引数に依存が現れる *)
functor UserService (Log : LOGGER) = struct
  fun create name = Log.log ("created: " ^ name)
end
```

SML の意味論的には

```
structure ConsoleLogger : LOGGER = struct
  fun log s = print (s ^ "\n")
end

structure Service = UserService(ConsoleLogger)
```

静的意味論 (Elaboration)

ConsoleLogger が LOGGER を
満たすか検証＋型名の生成

動的意味論 (Evaluation)

モジュール本体のコードを実行
型の解決は行わない

「実行時にシグネチャが合わなかった」という失敗の規則は **存在しない**

生成性

`:>` は **封印のたびに** 新しい型名を生む：

```
structure M = struct type t = string end

signature SIG = sig type t end
```

```
structure A :> SIG = M
```

⬇

A.t = t_1

```
structure B :> SIG = M
```

⬇

B.t = t_2

$t_1 \neq t_2$ —— 封印のたびに $T' \cap (T \text{ of } B) = \varnothing$ (Rule 53)

生成的ファンクタ

```
signature DICT = sig
  type key
  type 'a dict          (* 抽象型 *)
end

functor Dict (Key : sig type t end)
  :> DICT where type key = Key.t = struct
  type key = Key.t
  type 'a dict = (key * 'a) list
end

structure A = Dict (struct type t = string end)
structure B = Dict (struct type t = string end)

(* A.dict ≠ B.dict / A.key = B.key = string *)
```

可変参照があるときに嬉しい

```
functor Cache () :> sig
  type token
  val save : string -> token
  val load : token -> string
end = struct
  val tbl = ref ([] : string list)   (* インスタンスごとの内部状態 *)
  type token = int                   (* tbl の添字 *)
  fun save s = (tbl := !tbl @ [s]; length (!tbl) - 1)
  fun load t = List.nth (!tbl, t)
end
structure A = Cache ()   structure B = Cache ()
```

`B.load (A.save "x")` で存在しない添字を引くとまずい

Fresh Name 生成により `A.token` ≠ `B.token` の **型エラー** で止まってくれる

補足：OCaml のファンクタは applicative

```
module MakeDict (Key : ORDERED) : sig
  type 'a t
  val insert : Key.t -> 'a -> 'a t -> 'a t
end = struct
  type 'a t = (Key.t * 'a) list
  let insert k v d = (k, v) :: d
end

module DictA = MakeDict(StringKey)
module DictB = MakeDict(StringKey)
(* DictA.t = MakeDict(StringKey).t = DictB.t ── 同じパスは同じ型 *)
```

同じファンクタに同じ引数なら同じ型

仮引数に ( ) をつけると generative を選べる (パスを隠す)

可変状態の対処

多相性と可変状態

SML は非純粋で可変セルを作れる

```
(* もし ref [] が 'a list ref に一般化されたら... *)
val r = ref []
val _ = r := [1]                    (* int を格納 *)
val s : string = hd (!r)            (* string として取り出し Σ(･ω･ ;) *)
```

ref が生成 ／ ! が読み出し ／ := が書き込み

多相性と可変状態の組み合わせは，型安全性を壊しかねない！

各言語のアプローチ

多相性と可変状態を同居させるには，どこかに静的な制限が要る：

Haskell　`ref` (IORef) の生成そのものを `IO` の中に閉じ込め

Rust　`let` 束縛をそもそも多相化しない (多相は fn のジェネリクスだけ)

SML　可変状態は許容しつつも **多相化する式の側を制限した**

値制約 (Value Restriction)

構文的に「値」である式 (non-expansive) だけを多相化する

(Section 4.7, p.21)

nexp ::= scon | [op] longvid | { [nexprow] } | (nexp)
| conexp nexp (conexp は `ref` を除く) | nexp : ty | `fn` match

BNF に載っているかを見るだけで **型の解析はいらない**

多相化の具体例

本当に安全かではなく，あくまでも構文的に判断

```
(* non-expansive *)
fn x => x                (* 'a -> 'a  *)
[]                       (* 'a list   *)
NONE                     (* 'a option *)

(* expansive *)
ref []                   (* 危険（弾いて正解） *)
compose (id, id)         (* 安全（巻き添え...）*)
```

イータ展開

巻き添えを食った式を `fn` で包む：

```
fun compose (f, g) x = f (g x)
fun id x = x

val f_ng = compose (id, id)                (* NG *)
val f_ok = fn x => compose (id, id) x      (* OK *)
```

- 実在コード 20 万行超を Wright が調査
- 値制約の導入で書き換え必要だったのはイータ展開 31 箇所ほか少数……

この計測の上で **規則の単純さと型安全性** を優先した判断

おわりに

ふりかえり

意味論の分離	型の情報に評価規則が依存しない
不透明シグネチャ	Fresh Name が型の同一性を切って内部表現を守る
生成的ファンクタ	適用ごとの Fresh Name がインスタンスの混入を止める
値制約	多相化を制限し，可変状態でも型安全性を保つ

どう実装するかではなく **どう振る舞うべきか** を固定

多種多様な実装

SML/NJ … CPS 変換，スタックを使わずにヒープ割り当て + GC
MLton … モジュールを跨いだ最適化，C と同等の実行速度 (AtCorder でも使える)
ML Kit … リージョン推論が主軸のメモリ管理 (GC 併用)
Poly/ML … Isabelle (証明支援系) の基盤，マルチスレッド
HaMLet … The Definition に忠実なリファレンス実装
LunarML … Lua / JS ターゲット

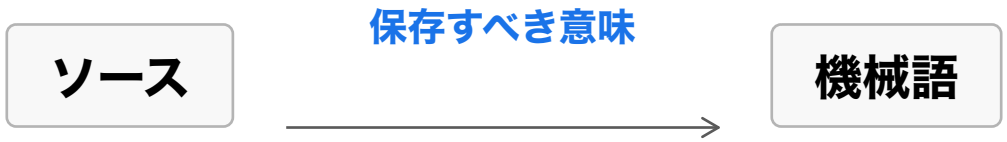
そのほか **CakeML**, **Moscow ML**, **Alice ML** など……

仕様外の機能を足そうとする動き：

SML/NJ … higher-order functor などの独自拡張
SML# … レコード多相 ／ C FFI ／ SQL 統合
Successor ML … SML '97 の保守的拡張の提案
1ML … モジュールとコア言語の統合

コンパイラへの信頼

普通はコンパイラの出力 (機械語) を毎回読んだりしませんが……



厳密さに差はあっても (C の未定義動作 ～ The Definition) 検証や議論の下地はある

SML の意味論を HOL4 上で機械化

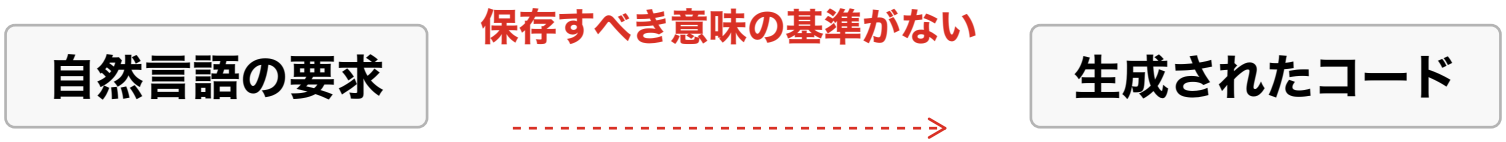
コンパイラの各パスが意味を保存することを証明した処理系

保存の定理を合成すると「このコンパイラは正しい」まで **定理** になる

意味論は The Definition のサブセットを独自に機械化

LLM の場合は？

何をもって正しい出力とするかはプロンプトを書いた人間の意図に依存する



生成されたコード自体には，型検査や lint や静的解析は掛けられる

これは断片的な性質で，自然言語の要求そのものと実装との対応関係を見ているわけではない

境界を設計する

要求との対応を見たければ **要求を検査できる形** に書き下ろすしかない

例えばユニットテストなら……

ソート　「出力が整列されている」だけなら，空リストしか返さない実装も通る
置換していることや安定性などを足していくと，テストは仕様に近づく

送金バッチ　「エラーなく完了」するだけでなく「処理の前後で残高の総和が不変」
振る舞いの書き下しの精度が高いほど仕様として成立していく

The Definition は，厳密な議論に耐えうるように，推論規則で言語を定めた
LLM の出力も，機械が検査できる不変条件でガードしていきたい

プログラミング言語の発展に要るものとは

処理系が事実上の仕様なら，処理系の保守が言語の進展に繋がる (Python, Ruby, OCaml)

一方で，文書が仕様なら，改訂が言語の進展に繋がるが……

ECMA-262 は TC39 が毎年改訂している

The Definition の改訂は ‘97 のあと出ていない (議論はあったが体制は残らず)：

> … we will not accept further revision of Standard ML. … We expect, however, the designers of any successor language to make it clear that their language is not Standard ML.
>
> 今後の改訂は受け入れない
> 後継言語の登場は自然だが，それが SML ではないと明示してくれ
>
> —— Milner & Tofte, sml-implementors ML (2001)

WebAssembly

The Definition (1997)

- 推論規則で言語を定義
- 型安全性の証明は後続の研究
- 改訂は止まった

WebAssembly (2017)

- 推論規則が W3C 仕様の本文
- 健全性は公開前に機械検証 (Watt 2018)
- W3C の作業部会が改訂を継続

The Definition が読めれば Wasm のコア仕様も読める (はず？)

$$\frac{C.\mathrm{locals}[x] = t}{C \vdash \mathrm{local.get}\, x : [] \to [t]}$$

—— WebAssembly Core Specification, Validation

(形式化を主導した Andreas Rossberg は HaMLet や 1ML の作者)

まとめ

いま OCaml が熱いらしいが……

(Standard) ML も熱い……！

Q & A

以下の質問には $O(1)$ で回答できます！

(ML の型推論は DEXPTIME-complete ですが [Mairson 1990])

- The Definition 自体にバグはないの？
- 標準ライブラリも推論規則で書かれているの？
- (Standard じゃない) ただの ML もあるの？
- SML ‘90 から ‘97 では何が変わったの？

もちろん，それ以外のご質問も歓迎です (*´˘`*)

参考文献

- Milner, R., Tofte, M., Harper, R., MacQueen, D. (1997) The Definition of Standard ML (Revised), MIT Press
- Kahn, G. (1987) “Natural Semantics”, STACS ‘87, LNCS 247
- Mairson, H.G. (1990) “Deciding ML typability is complete for deterministic exponential time”, POPL ‘90
- Milner, R. and Tofte, M. (1991) “Co-induction in Relational Semantics”, Theoretical Computer Science, 87(1)
- Wright, A.K. (1995) “Simple Imperative Polymorphism”, Lisp and Symbolic Computation, 8(4)
- Lee, D.K., Crary, K., Harper, R. (2007) “Towards a Mechanized Metatheory of Standard ML”, POPL ‘07
- Kumar, R. et al. (2014) “CakeML: A Verified Implementation of ML”, POPL ‘14
- Rossberg, A. (2015) “1ML – Core and Modules United”, ICFP ‘15
- Haas, A., Rossberg, A., Schuff, D.L. et al. (2017) “Bringing the Web up to Speed with WebAssembly”, PLDI ‘17
- Watt, C. (2018) “Mechanising and Verifying the WebAssembly Specification”, CPP ‘18
- MacQueen, D., Harper, R., Reppy, J. (2020) “The History of Standard ML”, Proc. ACM Program. Lang., 4 (HOPL)