ChatGPT

# JavaScript/TypeScriptにおける機械学習型型推論モデルの研究系譜

## はじめに

JavaScriptやPythonなどの動的型付け言語は人気が高い一方で、静的型付け言語に比べIDE支援や実行前のバグ検出が不十分という課題があります。JavaScriptを静的型付けに拡張したTypeScriptや、FacebookのFlowといった**逐次的型付け（gradual typing）**手法が登場し、既存の動的コードベースに部分的に型注釈を追加していくことが可能になりました [1]。しかし、大規模プロジェクトの型注釈移行には依然として開発者による手作業が必要で、Airbnbの事例では600万行のJavaScriptをTypeScriptに移行するのに2年以上かかったと報告されています。そこで、近年**機械学習 (ML)**を用いて未注釈コードから型情報を自動推定し、型付けを支援または自動化する研究が盛んになっています。

本レポートでは、JavaScript/TypeScriptにおける機械学習型の型推論モデルの系譜を概観します。まず、モデルのアーキテクチャ観点から**トークン列ベース・構文木 (AST)ベース・グラフベース**に分類し、それぞれの特徴を説明します。次に、代表的なモデル（DeepTyper、LambdaNet、NL2Type、Typilus、TypeWriter、OptTyper、InCoder、CodeT5+、Codexなど）の手法と成果を紹介し、TypeScript特有の型システム要素（ユーザ定義型、ジェネリクス、合同型・Union、関数オーバーロード等）への各モデルの対応について議論します。また、各モデルが扱う型ラベルの種類やその予測精度（Accuracy、Precision/Recall、F1など）の比較、ならびにモデル学習に用いられるデータセット（型付きTypeScriptコードコーパスやAST構造、Flowの型付きコードなど）の特徴にも触れます。さらに、大規模言語モデル（InCoderやCodexなど）と小規模モデル（DeepTyperやLambdaNetなど）の性能や適用シナリオの違いを比較し、型推論モデルの精度と**下流タスクへの影響**（コード補完、IDE支援、型移行の労力削減など）の関係について整理します。

特に近年の動向として、TypeWeaverやOpenTauといった研究で**「型推論の評価指標」を精度（正解型との一致率）から「移行後のコードが型チェックに通るか」という実運用上の成功指標へとシフトする動き**が見られます。本レポートではこの流れにも言及し、従来の静的評価から実用的な移行成功率評価への転換について解説します。図表として、モデル分類の概念図や代表モデル間の性能比較表、型の種類分類と各モデルの対応状況などを適宜示しながら、以上を総合的に日本語で報告します。

## 型推論モデルの分類: トークン列・AST・グラフ

機械学習を用いた型推論モデルは、コードの表現方法とモデル構造により大きく三つに分類できます。

- **トークン列ベース**: コードをテキストのシーケンス（トークン列）として扱い、自然言語処理に類似した系列モデルで型を予測するアプローチです。典型的には、ソースコード中の識別子や周辺の単語を順に読み込み、各識別子に対応する型を推定します。RNN（再帰型ニューラルネット）やTransformerなどの系列モデルが用いられ、機械翻訳のように「型なしコード」から「型注釈付きコード」への翻訳とみなす手法もあります。**DeepTyper** [2] はこの代表例で、JavaScriptをTypeScriptに変換する形で大量の対応データを生成し、シーケンス間翻訳モデルを訓練しました。また**NL2Type**はコード本体を無視し、関数名・パラメータ名・コメントといった**自然言語情報**の系列から関数の型シグネチャを推定する特殊な例です。近年の大規模言語モデル（LLM）も基本的にはコードをテキスト系列として扱って生成を行っており、このカテゴリに含まれます。

- **ASTベース**: ソースコードの抽象構文木（AST）の構造をモデルに組み込むアプローチです。コードを単なるトークン列ではなく**木構造**として入力し、構文木を横断する**木RNN/Tree-LSTM**や、ASTノードのシーケンス変換（例: 前順序やDFS順に列挙）によって学習を行います。これにより、ネストしたブロックやスコープ情報など構文的な文脈をより適切に捉えられます。例えば、あるモデルではASTノードの種類（関数宣言・変数参照・リテラル等）に応じ埋め込みベクトルを学習し、構文木を下位から上位へ伝搬するニューラルネットワークで型クラスを分類する、といった手法が考えられます。具体的な例として単独の有名モデルは少ないものの、Salesforceの**CodeT5+**シリーズでは事前学習に構文木情報を組み込む工夫をしており、TypeScriptの型推論タスクへの微調整にも応用されています（後述）。

- **グラフベース**: コード中の型に関する関係をグラフ構造で表現し、**グラフニューラルネットワーク(GNN)**で推論するアプローチです。これは近年注目された手法で、ASTよりも柔軟に**データフロー**や**型依存関係**を表現できます。例えばTypeScriptコードから**型依存グラフ**（Type Dependency Graph）を構築し、ノードを型変数（プログラム中の各変数や式に対応）、エッジを「二つの変数が加算される」「あるクラスのサブタイプ関係」等の制約や関係としてモデル化します。GNNを用いることで、グラフ上で情報（型の手掛かりとなるヒント）を伝搬させ、文脈的に一貫した型予測が可能になります。**LambdaNet**はこの手法をTypeScriptに初めて適用したモデルで、グラフノード間のメッセージパッシングにより未知のユーザ定義型も含めて予測できることを示しました。またPython向けの**Typilus**は、プログラムの構造・名前・パターンに基づくグラフ推論と**メタ学習**を組み合わせ、**一度も見たことのない型（オープンボキャブラリ）**でも1ショット学習で予測可能な点を特徴としています。この他、論理的な型制約グラフとニューラル予測を統合した**OptTyper**や、コードプロパティグラフにLLMの予測を統合する最近の**OpenTau/CodeTIDAL5**（後述）などもグラフベースの考え方を応用しています。

以上の三種はいずれも一長一短があります。トークン列モデルは大量のデータで汎用的に学習しやすい反面、**スコープや型文脈**を見落とすことがあります。ASTモデルは構文理解に優れますが、独自モデル構築が必要でユーザ定義型など**木に現れない情報**を直接扱いづらいです。グラフモデルは型システム知識を反映しやすいものの、グラフ生成に静的解析が要るため実装コストが高く、また大量のノード・エッジを処理する計算負荷もあります。そのため、近年の研究では**ハイブリッド**（論理的制約＋ニューラル、生成モデル＋検索など）なアプローチも模索されています。

## 主な機械学習型型推論モデル

続いて、JavaScript/TypeScriptの文脈で重要な機械学習型の型推論モデルを年代順に概観します。それぞれの手法と特徴、成果について述べます。

- **DeepTyper (FSE 2018)**: Hellendoornらによる、JavaScript/TypeScript向け初のディープラーニング型推論モデルです。TypeScriptで記述されたコードから型注釈を除去してJavaScriptとの**対訳コーパス**を自動生成し、これを使って**シーケンス変換モデル**（双方向RNN）を学習しました [2]。コードの字面上の特徴に加え、同一変数の複数箇所への予測に一貫性を持たせる**コンシステンシーレイヤ**を導入し、各識別子にもっともらしい型を割り当てます。評価では**トップ1正解率　約60%**、トップ5では80%以上と良好な結果を示し、既存の静的解析や従来手法（JSNice等）を大きく上回りました。またDeepTyperはTypeScriptコンパイラの**CheckJS**機能（型推論で得られない箇所を`any`とする部分的な型チェック）と組み合わせる実験も行い、信頼度の高い予測のみを型注釈として適用することで、**約95%の精度で4,000箇所以上の新たな型注釈を付加**できたと報告しています [2]。ただしDeepTyperは訓練データに出現しない**ユーザ定義型**（クラスや独自インターフェース名など）をそのまま予測することはできません。予測可能な型は訓練コーパス上の有限な辞書に限られ、未出の型名は`any`に置き換えるなどの処理をしていました。この制約や、同じ変数でも出現場所ごとに別々に予測することによる矛盾などが課題として残っています。

- **NL2Type（ICSE 2019）**: Pradelらによるアプローチで、**名前やコメントといった自然言語情報から型を当てる**というユニークな手法を提案しました。JavaScriptの関数定義約16万件に対し、JSDocコメントや関数名・引数名に含まれる単語をLSTMベースのニューラルネットに入力し、その関数の引数型や戻り値型を分類予測します。すなわち「実装コードは見ずに、名前付けやコメントから型を推測する」モデルです。例えば「isEnabled」という変数名から`boolean`型を、「getUserName」という関数名やコメントから戻り値を`string`型と推測するといった具合です。NL2Typeはトップ1予測で**適合率84.1%、再現率78.9%**を達成し、トップ5では95.5%/89.6%に向上しました。これは当時の最先端モデルJSNiceやDeepTyperをF1スコアでそれぞれ約29ポイント、約17ポイント上回る改善でした。特にNL2Typeは**コメントなし**の場合でも精度が大きく落ちない（F1で数ポイント低下するのみ）ことから、識別子名に含まれる暗黙の人間の意図をうまく学習していることが示唆されます。一方で実装コードを無視するため、例えば「実際には数値計算に使われるが名前にヒントがない変数」の型推定は苦手です。またDeepTyperとはアプローチが補完的であり、NL2Typeでのみ当たる予測も多く存在することが報告されています。

- **LambdaNet（ICLR 2020）**: Weiらによるモデルで、TypeScriptコードに対して**グラフニューラルネット（GNN）**を用いた確率的型推論を実現しました。LambdaNetはまず軽量な静的解析により各プログラムから**型依存グラフ（type dependency graph）**を構築します。このグラフではノードは型付けすべき各シンボル（変数や式）を表し、エッジは「ある変数の値が別の変数に代入される」「ある引数が関数のパラメータとして使われる」等、型間の制約関係を表現します。さらに識別子名やリテラル、演算の種類といった情報もノードに埋め込み特徴量として持たせます。GNNによるメッセージパッシングで、これらの関係に沿って型に関する情報がグラフ全体に伝播し、最終的に各ノードに確率分布として型予測結果が得られます。LambdaNetの大きな利点は、DeepTyperのような固定辞書に縛られず**ユーザ定義型**も予測できることです。例えば訓練時に見たことのないクラス`MyNetwork`でも、プログラム内でそれが生成され他の型とどのように使われているかという**コンテキスト**から推定できます。実験ではGitHub上の多数のTypeScriptプロジェクトで評価され、組み込み・標準ライブラリ型に限定したタスクで**トップ1精度75.6%**（DeepTyperは61.5%）、全ての型（ユーザ定義型含む）を含めた場合でも**トップ1精度64.2%、トップ5精度84.5%**という高い性能を示しました。これは当時のTypeScriptコンパイラの推論精度(約9%)を大きく上回り、機械学習による型推論の有用性を示す結果です。もっとも、LambdaNetはDeepTyperに比べモデルが複雑で、訓練にも推論にも静的解析が必要です。加えて、後述する評価観点では**安全すぎる予測**（例えば未知の型に対し`any`を予測するなど）は精度上は有利ですが型移行としては有益でない場合もあり、この点は次世代の課題となりました。

- **Typilus（PLDI 2020）**: Allamanisらの提案したモデルで、Pythonの動的型付けコードを対象にしたものですが、アプローチがLambdaNetと類似しているためここで紹介します。TypilusはGNNベースで**プログラムの構造・識別子名・コーディングパターン**を考慮した表現学習を行い、型推論を行います。最大の特徴は、型予測をクラス分類ではなく**メトリック学習**として定式化した点です。具体的には、あらゆる型を連続空間上の点（**TypeSpace**と呼ぶ埋め込み空間）にマップし、各シンボル（変数）の型的性質を表す埋め込みベクトルを学習します。学習後、新しい型については1つでも例があればその埋め込みを計算でき、空間的に近い既知の型と比較して推定を行えます。これにより**未知のユーザ定義型や極めて稀な型**でもワンショットで予測可能となり、オープンボキャブラリ問題を解決しています。TypilusはPython向けに評価され、予測に自信がある場合には全シンボルの70%に対して型を提案し（残り30%は`unknown`として無理に予測しない）、その提案の95%は実際に型チェックを通る正しい型であったと報告されています。またTypilusは既存の誤った型注釈の検出にも応用でき、実際に有名オープンソースプロジェクト（fairseqやALLENNLP）の型定義ミスを見つけ出し、開発者に受け入れられた事例も示されています。以上より、TypilusはLambdaNetと同様のGNNアプローチをさらに発展させ、型空間そのものを学習することでスケーラビリティを高めたものと位置づけられます。

- **OptTyper（2020）**: Pandiらの研究で、**論理と学習の融合**による確率的型推論フレームワークを提案しました。OptTyperはまずTypeScriptコードに対し従来型の型推論（コンパイラによる制約集計）を行い、型の取り得る選択肢に関する**論理制約**を抽出します。例えば「変数`x`は数値演算に使われている

からnumber型またはそのサブタイプしかあり得ない」等のハードな制約です。一方で、識別子名や出現パターンから得られる「推測上のヒント」を**自然な制約**と位置付け、こちらは不確実な情報としてディープラーニングモデルで型確率を予測します。OptTyperの肝は、これら論理制約と学習による確率分布とを組み合わせるために、型推論問題を**連続最適化問題**に緩和して解く点です。要は「論理制約に反しない範囲で、学習モデルの予測確率を最大化するタイプアサインを見つける」という一貫性の取れた解を計算します。この手法を用いてJavaScriptライブラリからTypeScriptの型定義ファイル(.d.ts)を生成する**OptTyperツール**を構築し、5800件の実プロジェクトで評価したところ、論理制約だけ・学習だけの場合に比べ誤った型推定が**50%少ない**高精度な結果を得られました。OptTyperはユーザ定義型も含め論理的に導ける型は積極的に採用でき、**論理（型システムの決定的情報）**と**経験知（ビッグコード統計に基づく予測）**を融合した先駆的な試みと言えます。

- **TypeWriter ( arXiv 2019/ICSE 2020 )**: Pradelらによる研究で、機械学習の型予測と**探索による検証(search-based validation)**を組み合わせた点が特徴です。TypeWriterはまずPartial Type Annotation予測器（実装はPython向けにLSTMベースで、関数の引数型と戻り値型を同時予測）を学習し、続いて予測結果を実際にコードへ反映して**型チェックをパスする組み合わせ**を探索します。具体的には、予測した型候補を関数単位で組み合わせ、Pythonの静的型チェッカ（FacebookのPyreを使用）でエラーが出ないか検証しながら、バックトラック検索で最適な注釈セットを見つけます。このようにモデルの確信度と型検査をフィードバックさせることで、単なる確率的予測以上に一貫性のある型付けが可能となります。Facebook社内の数百万行のコードベースおよびオープンソース1,137プロジェクトで評価した結果、TypeWriterの予測器単体でも戻り値型で**F1=0.64 （トップ5では0.79)**、引数型でF1=0.57（トップ5で0.80)と既存モデルを上回り、さらに探索による検証を組み合わせると**14%〜44%のファイルを完全自動で型付け可能**（型チェックエラーなく注釈挿入できた）という成果を収めました。このアプローチは「精度向上だけでなく実際に型移行をどこまで自動化できたか」を重視した点で画期的であり、実際FacebookではTypeWriterを開発者向けツールとして導入し、すでに何千件もの型注釈が人手の最小修正で受け入れられたと報告されています。TypeWriterはPython対象でしたが、「予測＋型チェック探索」の発想はTypeScriptなど他言語にも応用可能であり、後述のTypeWeaverやOpenTauの先駆けとなりました。

- **大規模言語モデルによる補完**:　2021年以降、OpenAIの**Codex**（GPT-3をコード用に微調整したモデル）やMetaの**InCoder**、Salesforceの**CodeT5+**、BigCodeプロジェクトの**SantaCoder**など、多数の大規模言語モデル(LLM)が登場しました。これらは数十億件規模のコードデータで事前学習され、高度なコード自動生成や補完能力を示しています。型推論においても、これらLLMは**暗黙的な知識**を活用してそれらしい型を補完できるため、新たなアプローチとして注目されています。例えば**InCoder (2022)**は6.7億パラメータのTransformerデコーダで、GitHubやStackOverflow由来の多言語コードに**穴埋め（fill-in-the-middle）**目的で訓練されました。InCoder自体は型予測専用ではありませんが、Maskトークンを使って「ここに型を入れる」プロンプトを与えると、それに適する型注釈を生成できます。TypeWeaverの研究ではDeepTyperやLambdaNetとともにInCoderを比較実験に用い、後述するとおり**最も高いファイル単位成功率**を示しました。OpenAIの**Codex**（12Bパラメータ）はGitHub上の公開コード全体で訓練されており、TypeScriptの補完も得意とされています。例えばGitHub Copilotとしてエディタに組み込まれ、関数の型注釈箇所でTab補完するとCodexが推測した型が挿入されるといった支援が既に利用可能です。Codexに関する学術評価は限定的ですが、InCoderと同程度かそれ以上の性能が期待されています。さらに**CodeT5+ (2023)**はオープンソースのコード向けLLM群で、Encoder-Decoder構造によりコード理解と生成の両タスクに柔軟に適応できます。ドイツの研究チームはCodeT5+を基に**CodeTIDAL5**というTypeScript型推論モデルを作成し、**型予測精度71.3%**を達成したと報告しています [3]。これは先行モデルより7.8ポイント高く、特にユーザ定義型の予測精度向上に寄与しています [3]。CodeTIDAL5は予測結果をコードプロパティグラフ解析ツールJoernに統合し、セキュリティ検査（データフロー解析）の精度を高める実用例も示されています [4]。このようにLLMは多様な文脈知識を持つため、これまで困難だったGenericsやUnion型など複雑な型表現も訓練データ中の類似パターンから推測できる可能性があります。ただしLLMはしばしば過剰な出力（存在しないパラメータの生成など）をするため、その**不適切な生成結果をフィルタする仕組み**（後述のTypeWeaverの処理など）が不可欠です。

以上が代表的なモデルの概要です。次節ではTypeScript特有の型システム要素に各モデルがどう対処しているかをまとめます。

## TypeScript特有の型機能への対応

TypeScriptには他言語にない高度な型システム機能が多数存在します。ユーザ定義クラス・インターフェース、**ジェネリクス（総称型）**、**ユニオン/インターセクション型**、関数の**オーバーロード**、**リテラル型**、**条件付き型**等です。機械学習モデルがこれらをどこまで扱えるかはモデル設計と訓練データに大きく依存します。

- **ユーザ定義型**: 開発者が定義したクラスや型エイリアスなど、未知の型名を扱うことです。DeepTyperやNL2Typeのような固定語彙に基づくモデルでは、訓練時に登場しない型名は正しく予測できません。例えば `MyClass` という新規クラスの型は `any` や既知の近い型に誤分類される恐れがあります。これに対し、LambdaNetやTypilus、OptTyperではプログラム内で使われているユーザ型の存在を前提に、名前や使用関係からその型名自体を出力できるようにしています。LambdaNetは型名を辞書分類ではなく**変数の数だけ出力ユニットを持つ回帰モデル**として扱い、未知の型でも「その変数が指すクラス宣言」を予測する形で対応しました [5] 。Typilusは前述のとおり型空間に埋め込むことで、新規型でも**ベクトル近傍から推測**できるようにしています。LLMの場合、事前学習に大量のプロジェクトが含まれるため有名なライブラリ型（例: Expressの `Request` 型等）は内部知識として持っている可能性があります。また、プロンプトにクラス定義本体が含まれていれば、その名前を返すことも容易です。このようにユーザ定義型については、近年のモデルはかなり対応力を上げてきました。実際、最新モデルのCodeTIDAL5は「型推論が最も必要とされるのはユーザ定義型の予測だ」と述べ、そこでの精度向上をアピールしています [3] 。

- **ジェネリクス（総称型）**: 型引数を持つクラスや関数、配列の要素型など可変長の型表現です。機械学習モデルが型名を当てる場合、例えば `List<int>` のような型は一つのラベルとして扱われます。DeepTyperやLambdaNetでも訓練データ中に `Array<string>` 等が頻出していれば一単語と見なして予測可能です。しかし、**型引数そのものを推論する**（例: 汎関数 `identity<T>` のTを推測する）のは、通常のアプローチでは困難です。TypeScriptコンパイラですら関数型の型引数は明示しなくても呼び出し箇所から推論しますが、MLモデル単体にそれを学習させるのはデータが膨大になります。OptTyperのようにコンパイラから制約をもらう手法なら、「この関数の戻り値と引数は同じ型Tである」等の関係を維持したまま予測できますが、OptTyper自体は総称型を明示的に出力する機能には踏み込んでいません。LLMは文脈から<>付きの型名も生成できますが、正しく文法に合ったジェネリック宣言を行う保証はありません。OpenTauでは、ツリー分解と探索を通じて親子関数の型を同時に予測する際、**型パラメータを子から親へ移植する**処理を入れています。これは、ある下位の要素で例えば `Array<number>` という型候補が出た場合、親の関数宣言部にジェネリック `<T>` を追加しパラメータ型と戻り値型を `T` に置換するといった試みです。もっとも、このような生成と型検査の組み合わせは計算量的に大変で、OpenTauでも候補を絞り込む工夫（組み合わせ数をPoisson分布でサンプリングなど）を凝らしています。総じて、ジェネリクスへの対応はまだ初期段階であり、多くのモデルは**具体的な型代入**（例: `<string>` つきの型）を当てるに留まり、ジェネリック型そのものを宣言・利用する提案は難しい状況です。

- **Union型・リテラル型**: TypeScriptでは `string|number` のように複数の型を許容するUnionや、特定の値だけ許すリテラル型（ `type A = 1|"one"` 等）があります。これらは型推論アルゴリズムにとって難易度が高く、MLモデルでもほとんど扱われていません。なぜなら、Union型は事実上無限の組み合わせがあり得るため、正解を一意に定めにくいからです。例えば、ある変数が文字列にも数値にも使われている場合、TypeScriptではUnionにするかanyにするかの判断が必要ですが、訓練データの文脈次第で答えは異なります。DeepTyperやLambdaNetは基本的に単一の型を予測対象としており、Union型は訓練中によく見られるもの（例えば `undefined|T` はオプショナルを表すので比較的出現）があれば予測ラベルに含まれる程度でした。OptTyperの論理制約アプローチなら、「複数候補が論理的に残る場合Union型を構成する」という拡張も考えられますが、論文内では言及されていません。

OpenTauでは型エラー数を減らす目的で、**ある型注釈候補がエラーを生む場合に自動で `any` に切り替える**処理が実装されています。この延長として、将来的には「代わりにユニオンにする」という選択も考えられますが、現在のところ自動生成されたUnion型の事例はほとんど報告されていません。リテラル型（例えば文字列リテラルのみを許す）はさらに予測困難で、訓練データにその型が明示されていなければモデルは `string` や `number` と推測するのが関の山でしょう。

- **オーバーロード**: 同じ関数名で引数の型組合せによって複数のシグネチャを持つ場合です。TypeScriptの `.d.ts` では典型的に関数オーバーロードが複数定義されますが、MLモデルが一度に複数のシグネチャを提案することはまずありません。OptTyperはライブラリの型定義生成を目指しましたが、得られた型はおおむね「ユニオン型を使った単一シグネチャ」か「最も一般的なケース1件」になると考えられます。TypeWeaverやOpenTauも、JSコードをTSに移行する際に複数シグネチャが必要になるケース（例えば関数内で引数の型による分岐がある）は、現状は**人手での修正**が必要な残されている難所です。モデルが自動で「2パターンのオーバーロードを出力する」能力はまだ実現されていません。

以上のように、TypeScript固有の高度な型機能の扱いは、ユーザ定義型についてはだいぶ克服されつつあるものの、それ以外（ジェネリクス、Union、オーバーロード等）はまだ十分とは言えません。特に**複雑な型の組合せ**を自動生成し整合性を取るには、探索と型検査の組み合わせなど計算的チャレンジが伴います。今後の研究課題として、この領域でのさらなる改良が期待されます。

## 型予測精度の比較と評価指標の変遷

各モデルの評価結果をまとめ、また評価指標の近年の変化について述べます。

### 従来モデルの精度比較

従来は「予測された型が人間の付けた正解注釈と一致するか」に基づく**精度 (Accuracy)**や**F1スコア**が主要な評価指標でした。表1に代表的なモデルの性能をまとめます。

【表1: 主な型推論モデルの性能比較 （トップ1精度やF1値）】

| モデル | 主な対象 | 手法 | ユーザ型対応 | 評価指標と結果 (トップ1) |
|---|---|---|---|---|
| DeepTyper (2018) | JS/TS | BiRNN + 一貫性 | ×（辞書内のみ） | Accuracy ~60%（トップ5で80%超） |
| NL2Type (2019) | JS (関数) | LSTM（名前・コメント） | △（推測のみ） | Precision 84.1%, Recall 78.9%（F1≈81%） |
| LambdaNet (2020) | TS | GNN (型依存グラフ) | ○ | Accuracy 64.2%（ユーザ型含む全体）<br/>Accuracy 75.6%（ライブラリ型のみ） |
| Typilus (2020) | Python | GNN + 埋め込み空間 | ◎ | 予測70%に自信あり・その95%が正当（型チェック通過） |
| OptTyper (2020) | JS (ライブラリ) | 論理制約 + DNN | ○ | 誤予測50%減少（既存比） |

| モデル | 主な対象 | 手法 | ユーザ型対応 | 評価指標と結果 (トップ1) |
|---|---|---|---|---|
| TypeWriter (2020) | Python | LSTM + 型検証探索 | △ | F1 = 0.64 (戻り値), 0.57 (引数) |
| CodeT5+/ CodeTIDAL5 (2023) | TS | T5系 Transformer | ◎ | Accuracy 71.3% [3] |
| InCoder (2022) | 多言語 | GPT-Dec (6B) | ◎ | （参考）Python戻り値推定 58.1% |
| Codex (2021) | 多言語 | GPT-Dec (12B) | ◎ | （非公開：HumanEvalで高性能、人間と同等の補完精度とされる） |

※DeepTyper精度は「非Any型への注釈付与箇所での正答率」、TypeWriterはトップ1予測のマクロ平均F1。
※Codexは公開精度データがないため記述的評価。

上記より、2018〜2020年頃はトップ1精度で50〜65%程度が一つの壁でしたが、その後の大規模モデルで飛躍が見られます。特にCodeT5+を用いたアプローチでは70%を超え、Codexのような超大規模モデルでは明示的評価はなくとも実用上かなり高精度な型補完ができると考えられます。もっとも、これらの数値は**開発者が記述した型注釈と一致した率**であり、複数の正解がありうる場面では過小評価となることがあります。例えば「推論的には問題ない別の型」をモデルが出力した場合でも、正解注釈と異なれば不正解とみなされます。この問題意識から、近年評価指標の見直しが提唱されています（次節）。

### 精度評価から型移行成功率評価へ

2023年のECOOPにおける**TypeWeaver**の研究は、型推論モデルの評価指標を抜本的に見直しました。彼らは「個々の注釈精度が高いことよりも、**自動移行ツールで付加した型によってコード全体が型チェックを通るか**を重視すべき」と主張しています。つまり、モデルの目的は単なる当てゲームではなく、**実際に使える型付けコードを自動生成すること**だからです。

TypeWeaverはDeepTyper・LambdaNet・InCoderという3種のモデルを組み合わせ可能なツールを構築し、513個のJavaScriptパッケージ（未型付けのnpm上位プロジェクトを含む）に対し、自動で型注釈を挿入してTypeScript化する大規模実験を行いました。評価指標として、**パッケージ全体がエラーなく型チェックを通った割合**（Package Success Rate）および**ファイルごとにエラーのない割合**（File Success Rate）を計測しています。結果は以下の通りでした。

- DeepTyperモデル使用時: **21%**のパッケージが完全移行成功、ファイルベースでは41%がエラーなし。
- LambdaNetモデル使用時: **9%**のパッケージ、約29%のファイルが成功。
- InCoderモデル使用時: **20%**のパッケージ、**69%**のファイルが成功。

もっとも**InCoder**が健闘し、ファイル単位では7割近くが自動型付け可能でした。一方、パッケージ単位で見ると3モデルとも2割以下に留まり、プロジェクト全体を完全自動移行するのは依然容易でないことが分かりました。興味深いのは、精度評価では最も優秀だったLambdaNetが、移行成功率ではDeepTyperより低かった点です。分析によると、LambdaNetは`any`のような**自明な型**を出さず具体的な型を狙う傾向が強く（結果として不整合な注釈が混入しエラーを生みやすい）、逆にDeepTyperは自信のない箇所では`any`を多用するため型チェック上はエラーになりにくい、という違いがありました。実際、エラーなく通ったファイル内の**トリビアルな注釈（`any`や`Object`等）の割合**はDeepTyperで約60%、LambdaNetで約25%、InCoderで約40%だったと報告されています。すなわち**安全策としてのany乱用**が多いモデルは型チェック成功率は上がる

が、そのままでは型移行の価値が薄れる、というジレンマが見えます。TypeWeaver論文では、このトレードオフを評価するため**trivial率**や**人間の書いた型との一致率**も合わせて分析しています。

TypeWeaverは加えて、移行後の型エラーの種類分析（トップ10エラーコードの頻度など）や、CommonJSモジュールをESMに変換することの効果検証、ケーススタディによる残存課題の検討も行っており、総合的に「現状どのような障壁で完全自動化が妨げられているか」を明らかにしました。例えば**外部ライブラリ型定義の欠如**（依存パッケージに型定義がないと、その型を参照する自作コードはエラーになる）や、**動的プロパティアクセス**（オブジェクトに存在しないプロパティへのアクセスは型上エラーになるが、実行時にはundefined許容で動いているケース）の扱いなどです。TypeWeaverでは、依存ライブラリを遡って型定義を導入したり、CommonJSの `module.exports` 形式をES6の `export` に機械的に変換する前処理を組み込むことで、これらの問題をかなり低減できると示しました。

TypeWeaver以降、この流れを継ぐ研究として**OpenTau　(2023)**があります。OpenTauはSantaCoderという1.1Bパラメータのオープンなモデル（LLM）をTypeScriptコード1,200万ファイルで**塗りつぶし型訓練**（Fill-in-the-Type; 型注釈箇所をマスクして当てさせる）し、さらにTypeWeaver同様の木構造分割と探索で大規模プロジェクトの段階的型付けを行いました。OpenTauはTypeWeaverより強力なモデルを用いたことで、**ファイル成功率47.4%**とTypeWeaver(InCoder)比で+14.5ポイントの改善を達成しています。一方、依然として1ファイルあたり平均3.3個の型エラーが残るなど完全自動には至っていません。OpenTauでも評価指標は**「何割のファイルがエラーゼロになったか」**であり、これはTypeWeaverの提唱した方向性を踏襲しています。またOpenTauは補助指標として**Typednessスコア**（プログラム全体で何%のシンボルに具体的型注釈が付いたか）を導入し、精密さと網羅性のバランスも評価しています。さらに興味深い点として、OpenTauは移行後コードのエラー数累積分布やエラーコード内訳も分析し、TypeWeaverとの比較で**約14.5%多くのファイルをエラーゼロにできた反面、Typedness（具体的型で埋められた率）はむしろ低下**したと報告しています。つまり、モデルがより `any` に頼った可能性を示唆しており、このあたりは今後の改善点とされています。

以上のように、**評価軸が「単一注釈の正解率」から「コード全体の型整合性（コンパイル通過率）」へ**シフトしつつあります。この流れは、研究をより実践的な価値に結びつける重要なステップです。ただし注意すべきは、型チェックを通すこと自体は `any` の氾濫でも達成できてしまうため、それを防ぎつつ有用な型情報を付与できたかを評価する工夫が必要という点です。TypeWeaverやOpenTauが**trivial型の割合**や**Typedness**を測っているのはこのためです。今後、現実のリファクタリング作業にどれだけ貢献したか、あるいは開発者が提案をどの程度受け入れたか（Human Acceptance率）といった評価も検討されるでしょう。

## トレーニングデータとコーパス

最後に、各モデルが学習に用いたデータとその特性について触れます。動的言語向けの型推論には、大規模な**型付きコードコーパス**の存在が不可欠です。幸いTypeScriptはオープンソースでも広く使われており、GitHub上に豊富なコードが存在します。Hellendoornらは2018年当時で既に数千のTypeScriptプロジェクトからペアデータを収集してDeepTyperの訓練に充てました [2]。NL2TypeはJSDocコメント付きのJavaScript関数162k件を集めています。LambdaNetはGitHubの実世界TSプロジェクトで訓練・評価を行い、そのテストセット規模は関数・変数合わせて数万件に及びます。近年、Jesseらは既存データセットを統合・拡充した**ManyTypes4TypeScript**を公開しました。これは13,953プロジェクト・539,571ファイルから成り、**型注釈数900万件超**という極めて大規模なコーパスです。従来のPython向けデータセット等と比べて約10倍の規模であり、シーケンスベース型推論モデルの訓練・評価に適したデータを提供しています。ManyTypes4TSはCodeXGLUEという機械学習ベンチマークにも採用され、例えばInCoder論文ではPython関数戻り値推定に同等のデータを使って58.1%精度を示すなど、比較評価に利用されています。

データの中身を見ると、型推論モデルに有用な情報が色々と含まれています。型注釈そのものはもちろん、**AST構造**（変数宣言か関数引数か等）、**識別子名のサブトークン**（ `user_id` なら["user","id"]）、**リテラル値**（ `0` や `""` から暗示される型）などです。モデルによっては訓練前にコードを前処理し、例えば**命名規則**を一般化することもあります。DeepTyperではレアな識別子を特殊トークンに置換し頻度調整するなどNLP的な

語彙管理をしています。LambdaNetやOptTyperは静的解析で得られた**制約グラフ**付きのデータを用意しなければならず、生成に一手間かかります。一方LLMの場合、事前学習でGitHub等の**あらゆるコード**を読んでいるため、型無しコードから間接的に型の知識を学習しているケースもあります（例えば「`push`というメソッド呼び出しがある→このオブジェクトは配列型らしい」など）。事実、Codexは型注釈の無いJavaScriptコードでも文脈から正しい型を推測することがあります。これは言語モデルが内部に獲得した**暗黙の型知識**と言えるでしょう。

またTypeScript以外にも、Facebook Flowの型付きJSコードや、C++/JavaのヘッダからJSライブラリの型情報を引き出す試みも考えられます。しかし主要な研究ではほぼTypeScript（または静的型付け言語を逐次的に利用しているPython）からデータを得ています。TypeWeaverはnpmトップ1000から型定義の無い513プロジェクトを抽出しましたが、その際**依存するnpmパッケージに型定義が存在するか**も考慮しデータセット化しています。これは、依存に型がないと移行時にエラーになるためで、逆に言えば**DefinitelyTyped**などコミュニティ提供の型定義集が移行成功率に寄与することを意味します。実際TypeWeaverは自動移行の最初のステップで、依存パッケージの型定義インストールを行っています。

まとめると、データの質と量はモデル性能の要であり、オープンソースの普及に伴い近年は非常にリッチなデータセットが利用可能です。今後はさらに実運用シナリオに合わせて、例えば「全く型のない大規模JSコード（ManyTypes4TSにはそういうファイルも含まれる）を段階的に型付けしていく過程」を再現するようなデータ分割や評価法も検討されるでしょう。また、生成モデルでは**プロンプト設計**も重要なデータの一部です。OpenTauはプログラムを関数単位に分割し、下位の推論結果を上位のプロンプトに差し込んで精度を高める工夫をしています。このようなデータとアルゴリズムのインタラクションも今後深化していくと考えられます。

## 大規模モデルと小規模モデルの比較・使い分け

大規模言語モデル(LLM)と従来の小規模モデルには、それぞれ利点と適用シナリオの違いがあります。LLM（数億〜数千億パラメータ級）は膨大な知識を内包し、文脈に応じた高度な推論が可能です。例えばCodexはAPIの仕様まで暗記しているため、型注釈だけでなく**関数の目的に沿った最適な型**を提案することすらあります。一方、小規模モデル（数百万パラメータ級の専用モデル）は目的特化型で、リソース消費が少なくデプロイの容易さや結果の説明可能性に優れます。

**性能面**では、近年のLLMは精度指標上も小規模モデルを凌駕しつつあります[3]。しかしTypeWeaverの結果が示すように、必ずしもLLMが全てにおいて勝っているわけではありません。LambdaNetのようなモデルは一貫性のある型を出そうとして型チェックエラーを誘発しましたが、逆にInCoderは多少的外れでも`any`に逃げるケースがあり、結果的に型チェック通過率が上がるという**皮肉な逆転現象**も起きています。これは、LLMが強力すぎて不要なコード（例えば存在しないパラメータ）まで生成してしまうリスクにも通じます。TypeWeaverではInCoderの出力を監視し、**余計なトークンを検出して棄却する**処理を入れていました。

**実用面**では、小規模モデルはコンパイラやエディタへの組み込みが容易です。DeepTyperやLambdaNetはオープンソース実装が公開されており、TypeScriptコンパイラ拡張として動かす試みも考えられます。実際DeepTyper作者らは型推論をエディタ補完に組み込む実験を行い、候補表示精度が向上したと述べています[6]。一方LLMは計算資源を要し、GitHub Copilotのようにクラウドサービス経由で提供する形が主流です。オフラインで巨大モデルを回すのは企業内利用でもない限り困難ですが、6B規模のInCoderならローカルGPUでも動かせるため、TypeWeaverはこれを採用しました。CodeT5+は220Mや770Mの中型も公開されており、これはLambdaNet程度のサイズなので組み込みやすいでしょう。

**役割分担**としては、現在の傾向では**小規模モデル＋システムで土台を作り、要所でLLMの知見を借りる**方向が考えられます。例えばOptTyperやTypeWriterは静的解析や探索でモデルを補完しましたが、これをLLMにも適用し、まずLLMでそれっぽい型を生成→コンパイラで検証→ダメなら別候補…という対話的なシステムも実現可能です。OpenTauはまさにLLMを候補生成器とし、型エラー数でランキングして最良を提示する構

成でした。一方、CodeTIDAL5のようにLLMを**静的解析ツールの強化**に用いるケースもあります [4]。Joernのデータフロー分析では型情報が乏しいと正確な攻撃経路追跡ができませんが、CodeTIDAL5で補完した型により多くの変数間の関係が把握でき、セキュリティ検知力が上がったと報告されています [4]。

総じて、LLMは「知識豊富なジェネラリスト」、従来モデルは「組み込みやすいスペシャリスト」と言えます。将来的には両者を組み合わせ、例えば**基盤モデルとしてLLMを使いつつ、その出力を小規模モデルや論理制約で整合性チェックする**ようなハイブリッド手法が主流になる可能性があります。また、TypeScriptコンパイラ自体に機械学習フックが入り、コンパイル時に統計モデルが補助する、といった発展も考えられます。現にMicrosoftはPythonの型推論にMachine Learningを組み込む試みをしているとの情報もあり、言語処理系とMLの境界が今後曖昧になっていく可能性があります。

## 型推論モデルの利点と下流タスクへの影響

最後に、機械学習型の型推論が開発現場や周辺ツールにもたらす効果について述べます。

**型予測精度と開発者体験**には正の相関があります。型注釈が自動で補完されれば、開発者はIDEからより適切な**コード補完**を受けられます。例えば、オブジェクトの型が分かればIDEはそのプロパティやメソッドを候補表示でき、コーディング速度と正確さが向上します。HellendoornらはDeepTyperを適用したハイブリッド補完で、提案精度が向上することを示しました [7]。また、静的な**バグ検出**も型があることで強化されます。Gaoらの研究によれば、JavaScriptの15%のバグは型注釈があれば防げたといいます [1]。ML型推論で自動付与した型でも、多くはコンパイル時エラー検出に役立つでしょう。もっとも、誤った型を付けてしまうと却って開発者を混乱させたり、不要なエラーを発生させるリスクもあります。従って、ツールとして提供する場合は**高い精度（特にPrecision）**が求められます。TypeWriterが実運用では自信度の高い型のみ提案し、90%閾値ではDeepTyperの誤った型を大幅に減らせたと報告したように [7]、信頼度指標と組み合わせて使うのが現実的です。

**型移行工数の削減**も大きなメリットです。TypeWeaverのようにパッケージ全体を半自動で移行できれば、開発チームは新機能開発と並行して型付けを進められます。Airbnbの事例では人手2年かかったものが、今後ツールで数ヶ月に短縮できるかもしれません。実際Facebookでは数千のPython関数にTypeWriterが注釈を加え、人間は微調整するだけで良かったとのことです。ただし完全自動はまだ精度的に難しく、人のレビューを前提とした**提案支援**という形が当面は現実的です。GitHub Copilotなどはまさに開発中にコメントから関数シグネチャまで自動生成しますが、開発者が都度確認・修正しながら使う想定です。同様に、自動型推論結果も**コードインテリジェンスの一部**として提示し、開発者が受け入れるか選択させるUIが考えられます。

**静的分析・リファクタリングへの貢献**も見逃せません。前述のように、セキュリティ解析では型があると「この関数の返り値はユーザ入力ではなく正規化済」といった情報が取り出せ、誤検知が減ります [4]。また、大規模コードの**モジュール依存解析**や**デッドコード検出**も型情報が支えとなります。これまでは人が苦労して型注釈を書いていた部分を、MLが補完することで、より多くのプロジェクトが高度な解析の恩恵を受けられるでしょう。

**課題**としては、モデルのバイアスや不足があります。例えば訓練データに現れない特殊なAPI型（DOM操作用型など）は提案されないかもしれません。実際NL2TypeはJavaScript独自の `DOMElement` 型を予測できた例がありますが、すべてのドメイン知識を網羅できるわけではありません。LLMであれば事前学習に由来する広範な知識が期待できますが、その分**幻覚**のような無関係な型提案も混ざりえます。これらに対処するため、将来的には**人間とのインタラクション**（例えば「この提案型は本当に受け入れますか？」と対話する）も視野に入るでしょう。

総じて、機械学習型の型推論モデルは年々精度と適用範囲を広げており、IDE支援や自動移行ツールとしての実用段階に近づいています。評価指標のシフトもその流れを後押しし、研究成果が実際の型付け作業削減に

直結するようになってきました。今後の展望としては、GenericsやUnion型などの完全対応、さらに精度向上と `any` 削減の両立、そして**開発者コミュニティへの受容**があります。型推論モデルが十分信頼できると認識されれば、言語仕様側で「型推論ヒント」を組み込む動き（例えば将来的TypeScriptコンパイラがオプションでML予測を参考にする等）も考えられます。機械学習とプログラミング言語の融合はまだ始まったばかりであり、JavaScript/TypeScriptはその最前線に位置しています。本レポートで概観した研究系譜が、今後のより安全で開発者フレンドリーな型システムの構築につながっていくことが期待されます。

【参考文献】


- V. J. Hellendoorn et al., "Deep Learning Type Inference," ESEC/FSE 2018 [2]
- Rabee M. et al., "NL2Type: Inferring JavaScript Function Types from Natural Language Information," ICSE 2019
- J. Wei et al., "LambdaNet: Probabilistic Type Inference using Graph Neural Networks," ICLR 2020
- M. Allamanis et al., "Typilus: Neural Type Hints," PLDI 2020
- I. V. Pandi et al., "OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints," 2020
- M. Pradel et al., "TypeWriter: Neural Type Prediction with Search-Based Validation," ICSE 2020
- M. Yee and A. Guha, "Do ML Models Produce TypeScript Types that Type Check?", ECOOP 2023
- F. Cassano et al., "Type Prediction with Program Decomposition and Fill-in-the-Type Training (OpenTau)," ArXiv 2023
- L. Seidel, "LLM-Powered Type Inference for Better Static Analysis (CodeTIDAL5 Talk Preview)," QwietAI Blog 2023 [3] [4]
- Kevin Jesse et al., "ManyTypes4TypeScript: A Comprehensive TS Dataset for Sequence-Based Type Inference," MSR 2022（データセット論文）


（以上）

---


[1] [2] Deep Learning Type Inference · Machine Learning for Big Code and Naturalness
https://ml4code.github.io/publications/hellendoorn2018deep/

[3] [4] Conference Talk Preview: LLM-Powered Type Inference for Better Static Application Security Testing - Preventing the Unpreventable | Qwiet[AI]
https://qwiet.ai/llm-powered-type-inference-for-better-static-application-security-testing/

[5] maruthg.com
https://maruthg.com/pubs/lambdanet.pdf

[6] [7] Deep Learning Type Inference
https://vhellendoorn.github.io/fse2018-j2t.pdf