ChatGPT

TypeWeaver: Learning Type Annotations from Examples 論文調査報告

論文の概要と目的・手法・貢献

TypeWeaverは、既存のJavaScriptコードをTypeScriptに自動移行（型付け）するために機械学習モデルを活用する研究です [1] [2]。従来、JavaScriptからTypeScriptへの移行（型付与）は大規模プロジェクトでは数年を要する手作業であり（Airbnbでは600万行の移行に2年以上 [3]）、これを自動化することが目的です。著者らは「個々の型予測の精度」が高いこと自体よりも、「自動移行ツールで挿入した型によってコード全体が**コンパイルエラーなくTypeScriptの型チェックを通るか**」を重視すべきだと主張しています [4] [5]。本研究では、自動型移行ツール**TypeWeaver**を開発し、機械学習モデルによる型予測結果を既存コードへ組み込んでTypeScript化する一連の工程を自動化しました [6]。TypeWeaverは以下のような前処理・統合手順を備えています [7] [8]:

- **依存パッケージの型情報導入**: 対象プロジェクトの依存関係も型定義（.d.tsファイルなど）を持つようにしなければなりません。そこで依存先を再帰的に型付けするか、型定義ファイルを導入します [7]。
- **モジュール形式の変換**: Node.js向けのCommonJSモジュールを使うコードは、そのままでは型情報が完全には活用できません。そこで必要に応じてコードをES6のECMAScriptモジュールに書き換えます [9]（※CommonJSのままでは型チェック上モジュール間の型情報が共有されないため [10]）。
- **型の織り込み（Type Weaving）**: 学習モデルは変数や関数に対応する型ラベルを推定するものの、生のJavaScriptコードに型注釈を**埋め込む**ことはしません。TypeWeaverでは予測された型を元のJavaScriptの適切な位置に挿入し、TypeScriptコードを生成します [8]。
- **非型トークンの除去**: 特に大規模言語モデル(LLM)などの生成モデルは、有効な型注釈になっていない文字列を生成することがあります。そのため型として文法的に不正な予測は除去・修正します [11]。

このようにして出力された型付けコードに対し、TypeScriptコンパイラによる型チェックを実行し、自動移行の成否を評価します。論文の主な**貢献**は以下の通りです [12] [13]:

- **TypeWeaverツールの提案**: 任意の型予測モデルと組み合わせてJavaScriptプロジェクトの型移行を評価できるフレームワークを提供しました（Section 3）。
- **評価用データセットの構築**: npm上位1000パッケージから513件を抽出し、既存に型定義のない（これまで型予測の評価に使われていない）コードを多数含むベンチマークを用意しました（Section 4.1） [14]。
- **型チェック成功率の測定**: 3種類のモデルを用いて型移行した場合に**どの程度コードが型チェックを通るか**をパッケージ単位およびファイル単位で評価し、挿入された型注釈のうち「自明なもの」（any等）の割合や、人手で書かれた型との一致度も分析しました（Section 4.2） [15]。
- **エラー発生パターンの分析**: 型移行後に生じる典型的な型エラーの種類や原因を分類し、頻度の高いエラーコードを特定しました（Section 4.3） [16] [17]。
- **モジュール変換の効果検証**: CommonJSからESモジュールへの変換前後で移行結果を比較し、型チェック成功率や精度に与える影響を調べました（Section 4.4） [18] [19]。
- **ケーススタディの提示**: 4つの具体的なパッケージについて、移行モデルごとの失敗要因や残された課題を詳細に検証し、典型的な難所を示しました（Section 4.5） [20]。

「高い精度 ≠ 移行成功」である理由と根拠

論文が強調するポイントは、「個々の型予測精度が高いこと」と「プロジェクト全体の型移行が成功すること」は必ずしも同じではないという点です [4] [5]。従来研究では、テストデータ上で予測された型が手書きの正解と一致する割合（Accuracy）を主な評価指標としていました。例えばDeepTyperやLambdaNetは、それぞれトップ1精度56.9%や64.2%（ユーザー定義型含む場合）を報告しています [21] [22]。LambdaNetはライブラリ型のみを対象とすれば75.6%という高精度も示されています [22]。一方で著者らは、TypeScriptへの実移行では「型チェックをパスできるか」が重要であり、**コンパイラが受理するか否かが真の正解**だと指摘します [5]。型移行ではプロジェクト内の**一箇所でも**不適切な型注釈があるとビルド全体が失敗してしまうため、精度評価で例えば90%正しく型付けできていても、残り10%の誤りが各所に散在すればプロジェクトとしては**型エラーが残存する**ことになります [23]。つまり、**ごく一部の予測ミスがあるだけで移行結果は使い物にならない**場合があるのです [5]。

さらにAccuracy指標は、「厳密な一致」だけを正解とみなすため誤解を招く可能性があります。TypeScriptでは型の互換性に幅があり、たとえば予測が手書きの正解と異なっていても、`any`型や汎用的な`Function`型であれば**一応コンパイルは通る**場合があります [24]。あるいは`string|number`というUnion型の予測が、正解では順序違いの`number|string`であっても、コンパイラ的には等価なので受け入れられます [24]。Accuracy評価ではこうしたケースは不正解と扱われますが、実際の移行作業では問題にならないケースです。一方でAccuracyが高くても、**型システム上受理されない組み合わせ**を予測してしまえば移行は失敗します。このため、著者らは**「高精度だから十分」という楽観は誤り**であり、自動移行ツールの評価には「コード全体が型チェックを通るか」を据えるべきだと論じています [4] [25]。

この主張の根拠として、TypeWeaverによる実証結果が示されています。3つのモデルの組み合わせで**最も性能が良かった場合でも、全ファイルが型チェックを通ったパッケージは全体の21%に過ぎなかった**のです [26]。すなわち**約5分の4のプロジェクトでは何らかの型エラーが残り、完全移行に至らない**という結果でした。これらのモデルはいずれも既存研究で高いAccuracyを報告していたにもかかわらず、プロジェクト全体を動かす観点では成功率が大幅に低下することが示されています。特にGraph Neural Networkを用いたLambdaNetは型予測精度ではDeepTyperより高い値を達成していたのに [22]、本研究における**パッケージ型移行成功率は9%程度と最も低く**なりました [27] [28]。一方、RNNベースのDeepTyperはAccuracyではLambdaNetに劣るものの、**より多くのパッケージで型チェックが通った（約20%）**という逆転現象も観察されています [27] [28]。これはDeepTyperが後述するように`any`等の安易な型を多用する傾向があり、一部の厳密さを犠牲にしてもコンパイルエラーを減らした可能性があります [29]。つまり**「精度が高いモデル」≠「移行結果が実用的なモデル」**であることが、定量的な実験によって示されたのです。

また、著者らは移行結果にエラーが大量発生する状態では**現場の開発者がツールの利用を諦めてしまう**恐れも指摘しています [30]。例えば自動付与した型によって何十箇所もエラーが出たら、その原因を一つ一つ潰すよりツールの提案を最初から無視するほうが早いかもしれません [31]。さらにTypeScriptコンパイラのエラーメッセージは、必ずしも間違った型注釈そのものを指摘するとは限らず、エラーが発生した式の位置を報告するため、**どの注釈を修正すればよいか自動で特定するのは難しい**場合があります [30]。このような理由からも、「精度が高いモデル」の出力をそのまま使うだけでは現実的な型移行支援にならないことが強調されています。

評価実験の内容（データセット・手法・指標）

データセット: 評価には上述の513個のJavaScriptパッケージが用いられました [14]。これらはnpmでダウンロード数上位の人気プロジェクトから抽出され、TypeScriptへの移行履歴がない純粋なJavaScriptも多く含まれます [32]。パッケージは以下のカテゴリに分類され、分析時に区別されています [33] [34]:

- DefinitelyTyped（型定義あり）: 既にDefinitelyTypedリポジトリ等で型定義ファイル(.d.ts)が提供されているライブラリ。さらに依存パッケージが無い場合と有る場合に細分。
- Never typed（型定義なし）: 型定義が存在せず、過去に型予測モデルの評価に使われていないパッケージ [32]。こちらも依存なし/ありで区別。

評価ではまず各パッケージに対し、TypeWeaverツールで自動型付けを実行しました。DeepTyper、LambdaNet、InCoderという特徴の異なる3モデルをそれぞれ用い、その出力（型を挿入したTypeScriptコード）をTypeScriptコンパイラで型チェックします [35] [36]。型チェックの結果に基づき、以下の指標が計測されています。

- **パッケージ単位の成功率**: パッケージ内の**全ファイルが型チェックエラーなく通った場合**にそのパッケージを「移行成功」とみなし、全パッケージ中の割合を算出しました [37] [38]（注: ツール実行自体が失敗したケースは分母から除外）。この基準だと、一箇所でもエラーが残れば失敗になります。
- **ファイル単位のエラーなし率**: プロジェクト全体ではエラーが残っても、**個々のファイルに注目**すると正しく型付けできている部分も多いことが期待されます。そのため全ファイル中でエラーのないファイルの割合も測定しました [39]。こちらはより細粒度で、部分的な成功度合いを示す指標です。
- **トリビアルな型注釈の割合**: 型エラーが出なかったコード中に挿入された型注釈のうち、`any`や`any[]`、汎用`Function`といった**意味的に情報量の乏しい型**が占める比率を分析しました [40]。これは「型チェックの通しやすさ」の裏返しとして、せっかく自動付与した型が有用でないものばかりでは本末転倒であるためです。
- **人手の型定義との一致度（精度）**: データセット中「DefinitelyTyped」に属するパッケージでは、本来の正解となる型定義（.d.ts）が存在します。そこでコンパイル後に生成した型宣言（d.ts）を取得し、関数シグネチャ（引数型・戻り値型）について**手書き定義と一致しているか**を計測しました [33] [41]。正解が`any`の場合はスキップし、型名がテキストで完全一致した割合を**Accuracy**（精度）として報告しています [42] [43]。
- **エラー発生状況の分析**: コンパイル時に発生したエラーの種類と件数を全モデル横断で集計し、特に頻出するエラーコードトップ10をまとめました（Table 5） [17]。また、エラーの出たファイル数分布や、CommonJS→ES6モジュール変換前後でのエラー変化も調べています [44] [18]。

主な結果: パッケージ単位では、前述の通り**エラーなく型付け完了できたのは全体の約21%（InCoder使用時）**に留まりました [26]。DeepTyperやInCoderではほぼ同程度（約20%）の成功率でしたが、LambdaNetは**わずか9%程度**と他より低く、依存関係のある複雑なパッケージほど成功率が下がる傾向も確認されています [27]。著者らは「この成功率自体は非常に低く残念に見えるが、パッケージ全体が完全に通るというのは非常に厳しい基準だ」と述べています [37] [45]。そこでより実態を反映するため**ファイル単位**で見ると、**エラーなしのファイルは全体の69%**にも達していました（InCoder使用時） [46]。つまり多くのファイルは自動型付けで問題なくコンパイルできており、エラーは一部のファイルに集中していることが示唆されます [47]。この結果から、開発者は移行ツールを使うことで**全ファイルではなく問題のある少数のファイル**に注力すればよくなる、という前向きな示唆も得られます [47]。

3モデルの比較では、DeepTyperとInCoderが概ね同程度の成果を上げ、LambdaNetは低めでした [48]。特に依存パッケージを多く含むケースではLambdaNetの成功率低下が顕著で、依存関係を持たない単独のパッケージの方がどのモデルでも移行が成功しやすい傾向でした [49] [27]。ファイルレベルでもInCoderは他モデルを上回り、全ファイルのうち約69%がエラーなし、DeepTyperとLambdaNetはそれぞれ43%と25%程度に留まります [29]。また「Trivial」な型の挿入割合を見ると、DeepTyperは**エラーの出なかったファイル中の約60%**もの注釈が`any/Function`等で占められており、LambdaNetは約25%とより厳密、InCoderは約40%で

中間的でした [40] [29]。これはDeepTyperが汎用型を多用してでもエラーを減らし、LambdaNetは比較的具体的な型を当てにいってエラーを増やしたことを物語っています [50] [51]。InCoderは両者のバランスを取り、適度に汎用型も使いながら最も多くのファイルをタイプチェック通過させています [50] [51]。

モデルの精度（人手の型との一致率）については、Dependencyの無いパッケージでは概ね高く、Dependencyありでは低下する傾向が報告されました [43]。全体的な**傾向としてLambdaNetのAccuracyが最も高く、DeepTyperが低い**という従来研究と同様のパターンが確認されています [43] [28]。にもかかわらず前述のように型チェック成功率では逆転するため、**Accuracy指標だけではモデル優劣を評価できない**ことが改めて示唆されます [28]。なおInCoderは汎用コード生成モデルでTypeScript専用ではないためAccuracy評価の直接の既存値はありませんが、本実験ではInCoderのAccuracyは中間程度（LambdaNetより低くDeepTyperより高い傾向）だったと読み取れます [18] [19]。

エラーの内容: 発生したコンパイルエラーの上位には以下のようなものがありました [17] [52]。

- **プロパティ不存在エラー (TS2339)**:「プロパティXは型Yに存在しません」。最も頻発したエラーで、特にLambdaNet出力では24,000件以上と突出しています [17]。オブジェクト型を誤推定したり、ある型に本来ないメンバをアクセスしている場合に出ます。LambdaNetは具体的なオブジェクト型を推測するぶん、正しくないプロパティ名でエラーになる率が高かったと考えられます。
- **型の不整合 (TS2322, TS2345)**:「型Xを型Yに割り当てられない」「引数の型Xはパラメータの型Yに割り当てられない」等、主に関数呼び出しや代入で型が合わないエラーです [17]。LambdaNetではこれも数千件と多発しました。例えば誤った型注釈のせいで関数の戻り値型が不一致になったり、パラメータ型が期待と異なると起こります。
- **未定義のシンボル/モジュール (TS2304, TS2307)**: 変数やモジュールが見つからないエラーです [16] [53]。これはモデルの型予測というより**環境構築上の問題**で、外部モジュールの型定義漏れ（依存先を型定義に含められなかった場合）や、`require`の動的ロードが上手くESモジュールに変換できなかった場合などに起こります [44] [54]。
- **関数の引数個数不一致 (TS2554)**: JavaScriptでは可変長引数をそのまま呼び出したり省略したりできますが、TypeScriptでは定義と個数が合わないとエラーになります [55]。LambdaNet出力で約1,200件観測されています [56]。
- **式が呼び出し不可 (TS2349)**: 関数でないものを呼ぼうとした場合のエラーです [56]。誤って関数型でない注釈を付けたのに関数として使っているケースなどが原因です。

これら以外にも、**TypeScript特有の制約**によるエラーが見られました。例えば「条件文の型が常に真/偽で無意味 (TS2367)」や「ジェネリック型に型引数が不足 (TS2314)」などです [57]。前者は常に真偽が決まる条件（型ガードの誤用など）で、後者はジェネリック型を予測したものの具体引数を付けられなかった場合に出ます（LambdaNetがジェネリック非対応ゆえによく出したエラー) [58] [59]。

モジュール変換の効果: 一部のパッケージについて、CommonJSモジュールのままでは`require`周りで型情報が欠落してエラーになっていましたが、ESモジュールに変換することで改善するケースがありました。例えば`regenerate-unicode-properties`というパッケージ（400以上のファイルから成る大型パッケージ）では、CommonJSのままだと全ファイルでエラーが発生しInCoderでのファイル成功率が11%に過ぎなかったのが、変換後は**89%のファイルがエラーなし**に跳ね上がったと報告されています [60]。一方でDeepTyperでは変換によってかえってエラーが増えるケースもあり、モデルによって良し悪しは分かれました [18] [19]。総じて「既にESモジュールを使っていたパッケージ」以外では劇的な改善例は少なく、変換前後で成功/失敗が入れ替わるパッケージも**ごく一部**に留まったとのことです [61] [54]。したがってモジュール形式の違いは成功率に影響するものの、本質的な課題は依然として型予測モデル側やコードパターン側にあると考えられます。

LLMベースの型予測・移行手法の限界と課題

本研究から、大規模言語モデル（LLM）等を用いた型予測アプローチの限界がいくつか指摘されています。

- **精度と実用性のギャップ**: 先述の通り、高いトップ1精度を誇るモデルであっても実際のプロジェクト移行ではエラーだらけになる場合があります [28]。特に**厳密な型**を狙うモデル（LambdaNetなど）は、Accuracyは高くとも少しの誤りが各所で発生し、結果として**全体の型チェックを阻害**してしまうことが分かりました [27] [28]。逆に**包括的すぎる型**（anyや汎用型）でごまかせばエラーは減りますが、それでは型移行する意義が薄れてしまいます [40] [29]。このトレードオフをうまく解決できていない点が現状のモデルの限界です。

- **トリビアルな型注釈乱用**: DeepTyperやInCoderの結果から、LLMベースのモデルは難しい箇所で `any` や `Function` といった**型チェックを通すだけなら都合の良い型**を出力する傾向があります [40]。これらは静的型検査上はエラーを抑制しますが、**型注釈としての価値（プログラマへの情報提供やバグ検出能力）は低い**ため、せっかく自動移行しても安全性や可読性の向上に寄与しません [40] [29]。著者らはこの問題を捉え、単に「型チェックを通すだけ」でなく**有用な型（non-trivial な型）を付与すること**が重要だと述べています [62] [31]。

- **無効な出力や文脈考慮漏れ**: LLMの出力する型注釈は常に構文的に正しいとは限らず、時に**TypeScriptの文法に合わない文字列**を生成します [63]。TypeWeaverではそれらを検出して削除・修正する処理を入れていますが [11]、このように**モデルの生出力をそのまま使えない**点は大きな課題です。また、LLMが学習データ上見慣れないライブラリやプロジェクト固有の文脈では不適切な型推定をする可能性もあります。本研究でも依存パッケージの型情報を先に与える工夫をしていますが、それでも未知の型に対しては対応しきれない場合があります（依存の有無で成功率が大きく変わったのはその表れです [49]）。

- **JavaScript特有のパターンへの対処**: JavaScriptには**逐次的に型を付けられないコードパターン**が存在し、LLMではそれを根本解決できません。例えば、図1(a)の関数 `f(x){ return x + x; }` は引数が数値なら加算、文字列なら連結し得るコードで、TypeScript的には `x` の型を `number| string` と定義できても、戻り値の型が入力次第で変わるため適切な注釈が難しい例です [64]。また図1(b)のように空オブジェクト `point` を作り後からプロパティ `x,y` を追加するコードも、TypeScriptでは `point` の型をあらかじめ宣言しなければエラーになります [65]。LLMがこれらのコードに型を“当てはめる”ことは困難であり、**人間がコードを書き換えないと解決できない**場合もあります [66]。著者らも「JavaScriptにはTypeScriptで筋の通った形にするには**書き換えが必要なイディオム**が存在する」ことをケーススタディから指摘しています [66]。実際、第4.5.4節では一つの変数を異なる型で使い回す例（後述）が紹介され、モデルでは対処不能なケースとして言及されています [67]。

- **コンパイルが通っても正しくない場合**: LLMベースの型付けは「コンパイルさえ通ればよい」という振る舞いをしがちですが、その結果、**型チェック上問題なくとも実行時に誤動作する**危険もあります [66]。論文では「移行後に型エラーはないが、それでも実行時エラーが発生する例」が実際に確認されています [66]。例えば、LambdaNetが `string` 型と予測した引数に対しコード中で `typeof` チェックで `function` を期待する処理があったケースでは、一見型エラーは出ないものの**型注釈が間違っているせいで実行時には必ず例外が投げられる**ことになりました [68] [69]（第4.5.3節の事例）。またInCoderの例では、ある関数の引数を `number` ではなく `any` で注釈したため型チェック上はOKでも、もし誤った型の値が渡されると検出できずに実行時エラーにつながりうる、という指摘もあります [70] [71]。このように**静的チェックをすり抜ける不適切な型**を与えてしまうリスクがLLMにはあり、精度や型チェック成功率だけでは測れない課題として残ります。

- **モデルの適用範囲の限界**: LambdaNetのようにモデル自体がサポートしていない型表現（例: ジェネリック型の具体引数）を出力してしまうケースも報告されています [58] [59]。このような**モデル固有**

の制約に起因するエラーは、現状では後処理で部分的に補正するしかありません（TypeWeaverではジェネリック型の一部簡易修正を実装） [72] 。また、InCoderのような汎用コード生成モデルを使うには入力プロンプトや生成位置の工夫（fill-in-the-middleを利用して型注釈のみ生成させるなど）が必要で、本研究では専用のフロントエンドを実装しています [73] [74] 。言い換えれば、**LLMをそのまま適用するだけでは不十分**で、課題に合わせたカスタマイズや補助処理が不可欠です。

以上のように、LLMベース手法には「型注釈を全自動で正確・有用に付与する」ための様々な壁が現状存在します。しかし著者らは、**完全自動でなくとも人間の移行作業を強力に支援できるツール**になる可能性は十分あると述べています [75] 。実験でも大半のファイルは自動で片付いたことから、残りの難しい部分だけ人手で直すアプローチでも生産性向上は期待できます [76] [75] 。今後の課題として、型チェックを通すだけでなく**実行時の振る舞いも評価する**ことや、多少誤った型でも受け入れる柔軟性、さらに多様なモデル（例えばGPT系）の活用も検討したいと述べられています [77] 。

論文における「移行の成功」の定義と評価方法

本研究で定義する「型移行の成功」とは、**自動付与した型入りのTypeScriptコードがコンパイルを通過すること**、すなわち**TypeScriptの型チェックに合格すること**です [5] 。JavaScriptにはもともと完全な「正解の型注釈」は存在しないため、「成功」の判断基準は**コンパイラが受理したか否か**のみが頼りになります [5] 。極端に言えば、TypeScriptコンパイラがエラーを報告しなければそれで移行は一応成功であり、逆に1つでもエラーが出ればその時点でプロジェクト全体としては未完了という扱いです [23] 。この基準は従来のAccuracy評価より厳しく、前述の通り1箇所の不適切な型注釈で全体が「失敗」と判定されてしまいます [23] 。

ただし、論文中の評価では上記の**パッケージ単位の成功率**に加えて、部分的な成功度を見るため**ファイル単位**のエラーなし率も利用しています [39] 。パッケージ全体ではエラーが残っていても、その中の何割のファイルが既に問題なく移行できたかを測ることで、移行作業の進捗や残作業量を評価できます [76] 。実際、InCoderでは**ファイルレベルでは69%がエラーなし**となり、パッケージレベル21%よりかなり高い値を示しました [26] [46] 。著者らはこれを「**細粒度の評価指標ながら有用**」と位置付け、移行ツールの実効性を測るもう一つの尺度としています [39] 。

また「成功」の質にも注目しており、型チェックさえ通れば良いというわけではない点は先述の通りです。エラーなく通っているファイルでも、その型注釈が `any` だらけでは有用性が低いため、**エラーなしかつ有用な型が付いているか**も評価しています [40] 。具体的にはエラーのなかったファイル内の注釈について、`any` や `Function` の割合を算出し、これが低いほど「質の高い成功」と見なせるとしています [40] 。さらに、成功したファイル・パッケージについて**本来期待される型と一致しているか**（精度）も測定し、単に通っただけではない妥当性の指標としています [33] [42] 。

総じて、論文内では「移行の成功」を多面的に捉えていますが、**基本的な定義は「型エラーが残っていないこと」**です [5] 。そこに「どれだけ自動でできたか」「付与された型は意味のあるものか」という観点を加味し、移行ツールの完成度を評価しています [40] [43] 。

精度が高くても移行が成功しない例（ケーススタディより）

論文内で紹介されている事例から、**予測精度が高く見えても移行に失敗するケース**をいくつか抜粋します。

- **ケース1: 型注釈の誤りが別の箇所のエラーとして現れる（decamelizeパッケージ）**
  `decamelize` は文字列をキャメルケースから小文字スネークケースに変換するライブラリで、第4.5.1節のケーススタディとして登場します [78] 。あるヘルパー関数 `handlePreserveConsecutiveUppercase` が2つの引数を取り（コード上ではArrow関数を変数に代

入）、他所から文字列を渡されて使われています [79] [80]。この関数、自体は開発者が型注釈を書いておらず、モデルが型を付ける必要があります。理想的には両引数とも文字列 `string` 型と注釈すべきところ、**DeepTyperは2つの引数を正しく `string` と予測できた**ものの、**肝心の関数自体の戻り値型を誤って `string` にしてしまった**ようです [78] [81]。その結果、関数を呼び出す側のコード（文字列を受け取ることを期待している箇所）で**「関数が呼び出せない/代入できない」というエラー**(TS2349)が発生しました [82]。TypeScriptのエラーメッセージは呼び出し元の行番号を指すため、一見「関数呼び出しの書き方が悪い」ように見えますが、実際の原因は**関数リテラルの型注釈が間違っており関数として扱われなくなっていた**ことでした [82]。このように、モデルの予測ミスが間接的に別の場所のエラーを誘発し、しかもエラーメッセージからは誤注釈箇所が特定しづらい例として紹介されています。DeepTyperの場合はパラメータ型は合っていたため**部分的には精度が高い予測**でしたが、**一箇所の不正確な注釈でファイル全体がコンパイルエラーになる**ことが分かります。

- **ケース2:「厳密すぎる型」が仇となる例（LambdaNetのFunction型乱用）**
  LambdaNetは推論した型が厳密すぎてエラーや不整合を生むケースも報告されています。先の `handlePreserveConsecutiveUppercase` 関数について、LambdaNetはこの関数自体の型を**汎用 `Function` 型**と予測し、引数の一つ `separator` を誤って `number` 型にしていました [80] [82]。一見ミスマッチに思えますが、TypeScriptでは `Function` 型は「引数・戻り値なんでもあり」のため、呼び出し側で文字列引数を渡しても**エラーになりません** [80] [82]。つまりLambdaNetは**実質anyに等しいFunction型**で包み込むことで型チェックエラーを回避してしまったのです。これは**移行ツール評価上は成功にカウントされる**ものの、注釈としては不適切です（separatorは本来stringであるべきなのに型チェック上は見逃される） [80]。この例では、LambdaNetはDeepTyperに比べ型推定精度が高いとされながら、誤った型（number）を含んでもエラーにならない注釈の付け方をしていたため、一種の「隠れた失敗」と言えます。**精度評価では関数型を当てただけで部分点になるかもしれませんが、移行という観点では誤注釈を見逃す危険な成功例**です。

- **ケース3: 型システム上のトリックによる見かけ上の成功（IEEE754パッケージ）**
  第4.5.2節では `ieee754` というパッケージの例が紹介されています。このパッケージにはバッファに浮動小数点数を書き込む関数があり、本来の型定義では引数に `buffer: Uint8Array, offset: number, isLE: boolean` 等が使われていました [83]。DeepTyperの出力では、あるバッファ関連の変数に対して**戻り値型の不一致エラー**(TS2322)が出たものの、それを修正すればコンパイル可能になるところまで漕ぎ着けました [84]。しかし**さらに見落とされた間違い**が潜んでいます。それはフラグ引数 `isLE` の型で、DeepTyperは本来 `boolean` であるべきこの引数を `number` と注釈してしまいました [85]。このミスは関数内部で `isLE` を真偽値として扱う処理がなかったため**コンパイラに検出されず**、結果として型チェック自体は通ってしまいました [85]。ところが、もし開発者がこの関数に誤った数値（0や1以外）を渡した場合、静的型は何も警告せず実行時に問題が起こりえます。つまり**「型チェック成功=正しい型が付いた」ではない実例**として、このケースが挙げられています。InCoderも同じ関数に対し、`buffer` に `Buffer` 型（Node.js組み込みでUint8Arrayのサブタイプ）を充てるなど一見もっともらしい注釈をしましたが、`value` 引数を `any` にしていたため、これも**エラーなく通るが安全性が下がった**例とされています [86] [87]。

- **ケース4: 一つの変数に二種類の型要求がある例（array-uniqueパッケージ）**
  第4.5.4節では、**同じ変数を異なる用途で使うために型の両立が不可能な例**が示されています [88] [67]。`array-unique` というパッケージからの抜粋で、一つの関数内に2種類のループがあります。`i` というループ変数を使い、まず**数値インデックスで配列を回すループ**(for-loop)が90行目に、続いて**オブジェクトのキーを列挙するループ**(for..in)が92行目にあります [67]。JavaScriptでは同じ変数 `i` を再利用できますが、TypeScriptでは前者では `i` は数値、後者では文字列であることが求められます [89]。LambdaNetは `i: number` と注釈しましたが、このままでは後半の `for..in` 部分で「stringを期待する場所にnumberがある」とエラーになります [89]。では `i: string` に変えれば良いかというと、今度は前半の数値ループがエラーになります [90]。「union型にすれば？」とも考えられますが、`for` 文のカウンタ変数に `string|number` は許されずコンパイラは依然エラーを出すでしょう。結

局、このケースは**一つの変数に二つの型役割を持たせているため、自動では整合する型付けが不可能**なのです [90]。TypeWeaverでの解決策としては、片方のループで別の変数名を使うよう**人間がコードをリファクタ**するか、変数宣言のスコープを分けるなどの書き換えが必要になります [91]。この例は**JavaScriptの柔軟さが静的型付けには相容れない典型例**であり、どんなにモデルの精度が高くても対応できない移行失敗のケースとして紹介されています [66]。

以上のケーススタディは、「あと一歩」で型移行が完成しない様々なパターンを示しています。モデルが大半の場所で正しい型を当てていても、一箇所の不一致でプロジェクト全体がエラーになる現象や、モデルがエラー回避のために不適切な汎用型を充ててしまう問題、JavaScript特有のコード構造による限界など、**精度の高さだけでは克服できない課題**が浮き彫りになっています [28] [66]。これらの具体例は、単純な精度指標では見えてこない移行実務上の難しさを如実に物語っています。

TypeBERT系モデル提案時における本論文の有用性（LLM手法の限界の文脈で）

TypeWeaverの知見は、近年提案されている**TypeBERT系のモデル**（BERTに基づく型予測モデル）やその他LLM活用手法を評価・議論する上で重要な示唆を与えます。本論文でも、最新の発展としてTypeBERTを用いたモデルである**DiverseTyper**に言及しており、これは組み込み型・ユーザ定義型の両方で最先端の精度を達成したとされています [92]。著者らはDiverseTyper/TypeBERT自体の評価は行っていませんが、そのような高精度モデルであっても**InCoder（汎用コード生成LLM）と近いアプローチ**であり、本研究で指摘した課題に直面する可能性が高いと述べています [92]。実際、TypeBERTに代表されるような事前学習済み大規模モデルは、単一の変数に対する型推定能力は非常に高い一方で、**プロジェクト全体の一貫した型整合性までは保証しない**ためです。本論文の結果は「高いAccuracyを報告するモデルほど注意が必要」なことを示しており、TypeBERT系モデルを提案・評価する際にも**型チェックを通す能力**や**トリビアル型の少なさ**など、従来軽視されがちだった観点を取り入れる必要性を教えてくれます [5] [40]。

例えば、新たなTypeBERTベースのモデルを開発する研究では、本論文を引用して「単純な精度向上だけでは不十分」である根拠にできます。TypeWeaverの実験から、LambdaNet（精度75%）よりDeepTyper（精度57%）の方が移行成功率で勝るという逆転現象が示されており [28]、これは**大型言語モデルによる高精度予測にも同様のリスクがある**ことを示唆します。TypeBERTのようなモデルを評価する際、本論文にならって**TypeScriptコンパイラでの検証を組み込む**ことで、より実用的な指標で性能を測れるでしょう。また、TypeWeaverが実装した依存関係の型取込みやモジュール変換、型織り込みといった工程 [7] [8] は、TypeBERT系モデルを実システムに統合する際にも避けて通れないステップです。これらの工程を怠ると、どんなにモデルが優秀でも現実のプロジェクトでは機能しない可能性があります。本論文はその点を体系立てて検証・報告しているため、TypeBERT系の研究者・開発者にとって**実践上のガイドライン**となります。

さらに、本論文は将来的な課題として「ランタイム挙動の評価」や「多少の誤りを許容する基準」についても言及しています [77]。TypeBERT系のモデル提案でも、単に型検査が通るコードを出力するだけでなく、**実行時の安全性**や**開発者が修正しやすい提案**といった観点を考慮すべきだという示唆が得られます。本論文で示されたケース（型チェックをすり抜ける誤注釈や、人手の介入が必要なパターン）は、TypeBERTのような高性能モデルでも無視できません [70] [67]。したがって、TypeBERT系の研究を発表する際には、本論文を引用しつつ**「我々のモデルはTypeWeaverで指摘された○○の問題に対処している/残課題である」**と議論することで、研究の位置付けや貢献を明確にできるでしょう。

総合すれば、**TypeWeaverの研究はLLMベース型推定手法の限界を客観的データで示した先駆的なもの**であり、TypeBERT系モデルの評価・改良を議論する上で非常に有用です。高精度モデルの落とし穴、エラー分布や典型パターン、そして評価指標の再考など、本論文の知見を踏まえることで、より実践的で信頼性の高い型移行手法の開発につなげられると考えられます。

参考文献: 本回答中で引用した論文「Do Machine Learning Models Produce TypeScript Types that Type Check?」 [1] （通称TypeWeaver論文）より適宜内容を参照しています。

---

[1] [2] [6] [26] (PDF) Do Machine Learning Models Produce TypeScript Types that Type Check?
https://www.researchgate.net/publication/368753897_Do_Machine_Learning_Models_Produce_TypeScript_Types_that_Type_Check

[3] [4] [5] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] Do Machine Learning Models Produce TypeScript Types That Type Check?
https://drops.dagstuhl.de/storage/00lipics/lipics-vol263-ecoop2023/LIPIcs.ECOOP.2023.37/LIPIcs.ECOOP.2023.37.pdf