ChatGPT

TypeWeaver: Learning Type Annotations from Examples (ECOOP 2023) – 研究内容の概要

1. 研究の全体的な流れと目的

TypeWeaverの目的とアプローチ: TypeWeaverは、未型付けのJavaScriptコードに対して機械学習モデルを用いて型注釈を自動付与し、TypeScriptへの移行（型移行）を支援するツールです [1]。既存研究では個々の型注釈予測の高い精度が報告されていましたが、著者らは**「型注釈の予測精度だけでは不十分であり、コード全体がTypeScriptの型検査を通るかどうか（type checkをパスするか）を評価すべきだ」**と主張しています [1] [2]。TypeWeaverのゴールは、モデルが予測した型をコードに織り込み、**コンパイル時の型検査エラーが出ないTypeScriptコードを生成すること**にあります。その理由は、たとえ個々の予測精度が高くても、わずか一箇所でも不適切な型注釈があるとパッケージ全体が型エラーで失敗し、開発者が大量のエラー対応に追われてツール自体を断念してしまう恐れがあるためです [3] [4]。したがって、本研究では**型検査を無エラーで通過すること**を型移行の成功指標と位置付け、その達成に注力しています [2]。

TypeWeaverの構成と処理ステップ: TypeWeaverは任意の型予測モデルの出力を受け取り、以下のステップでJavaScriptコードをTypeScriptコードへと変換します [5] [6]（型織り込み; type weaving）：

1. **依存パッケージの型定義導入:** 移行対象プロジェクトの依存関係について、型定義ファイル（`.d.ts`）が存在するかを確認し、なければ取得します。これにより外部依存の型情報をプロジェクトに取り込み、型検査で参照可能にします [7]。依存先に型定義がない場合には、その依存も再帰的に型移行する必要がありますが、本研究の評価対象では全ての依存パッケージに型定義が存在するようフィルタリングされています（後述） [7]。

2. **モジュール構文の変換 (CommonJS→ESM):** Node.js向けに書かれたJSコードはCommonJSの`require/module.exports`形式を使用している場合があります。一方TypeScriptではECMAScriptモジュール(ESM)形式（`import/export`）でないと型情報を完全には保持できません [8]。そこで、CommonJSモジュールを検出して可能な限りESM構文へリファクタリングします [8]。この変換により、モジュール間のインターフェース型も静的に解釈され、型検査を確実に適用できるようにします。

3. **型注釈の織り込み (Type Weaving):** 機械学習モデルが各変数や関数に対して予測した型を、元のJavaScriptソースコードに挿入してTypeScriptの型注釈付きコードにします [9]。モデルは通常コード中の各シンボルに型ラベルを与えるだけなので、TypeWeaverがソースコードに適切に型注釈を書き込む処理を自動化します [9]。

4. **不正確な予測の除去:** 自然言語処理モデル（例: InCoder）の出力する型注釈候補には、TypeScriptの型構文として不正なトークン列（存在しない型名や構文エラーとなる文字列）が含まれることがあります [10]。TypeWeaverはそのような「型ではない予測」を検出して除去・修正し、最終的に得られるコードが構文的にも正しいTypeScriptになるようにします [10]。

以上の工程を経てTypeWeaverは変換後のコードをTypeScriptコンパイラでコンパイルし、**型エラーの有無を確認**します [11]。TypeWeaver自体は型予測モデルに依存しない設計で、任意のモデルをプラグインのように差し替えて利用できるフレームワークとなっています [12]。

2. 使用された評価用データセット

データセット概要: 評価にはnpmパッケージの実コードからなる大規模データセットが用いられました。著者らはまずnpm公式レジストリでダウンロード数上位1000のパッケージ（2021年8月時点）を出発点とし、TypeScript型移行の評価に適した513個のJavaScriptパッケージを抽出・整備しています [13] [14]。このデータセット作成にあたり、以下のフィルタリングとクレンジングが行われました [13]:

- **依存パッケージの補完:** 上位1000中のパッケージがさらに他のパッケージに依存している場合、それら依存もデータセットに追加しました（「transitive dependencies」も含めクローズドな集合にする） [13]。加えて、各依存パッケージについてTypeScriptの型定義ファイル（@types/パッケージ など）が存在するもののみ残し、存在しない場合は除外しました [7]。この操作により、**データセット中のすべてのパッケージは外部依存の型情報が利用可能**となっています [7]。

- **ソースコードの取得と不要物除去:** npmに公開されているバンドル済み・圧縮済みコードではなく**元のソースリポジトリからコードを取得**し、リポジトリへのURLが無い・削除済みなどで入手できないパッケージは除外しました [15]。さらに、Mono-repo（1つのリポジトリで複数パッケージを管理）由来で重複を含むもの（例: Babelプロジェクトが100以上の個別パッケージに分かれるが実質同一コードを多重取得してしまうケース）は重複部分を除外しています [16]。テスト専用のコード（`/test` ディレクトリ等）も型移行の対象から外し、本評価の範囲を実装コードに限定しました（テストコードへの型付与は対象外）。

- **大規模・不適切なパッケージの除外:** 極端に大きなパッケージ（例えばJavaScriptで書かれていない、あるいはコード行数が異常に多いもの）は評価の現実性を考慮し除外しています。また一部「ビルドされたコード（例えばminify済みJSなど）」も対象から外し、あくまで**人間がメンテナンスするソースコード**として妥当な範囲のものを残しています [15]。

カテゴリ分類とパッケージ数: クレンジング後の513個のパッケージは、以下の2軸で分類されています [17]:

- **型定義の有無:** パッケージ自体に既存のTypeScript型定義が存在するか（具体的にはDefinitelyTypedリポジトリに型定義が用意されているか、またはパッケージ作者提供の型定義が含まれるか）で区分します [14]。既に型定義が存在するものは「DefinitelyTyped」カテゴリと呼び、**これまで一度も型付けされたことのない**パッケージは「Never Typed」カテゴリと呼びます [14]。後者は過去研究でも扱われてこなかった生のJavaScriptで、TypeWeaverによる初の型付け対象となります。

- **依存パッケージの有無:** 外部依存関係を持たないスタンドアロンなパッケージか（No Deps）、他のパッケージを依存に含むか（With Deps）で区分します。依存がある場合でも上述のとおり**その依存先には型定義が存在**することが保証されています。

これら2軸の組み合わせでデータセット内のパッケージは4つのカテゴリに分類されます。それぞれの件数は**表1**にまとめられています [18] [17]。

- **DefinitelyTyped, no deps:** 型定義あり・依存なしのパッケージ数は **286個**（計2,692ファイル, 約12.3万行） [18]。
- **DefinitelyTyped, with deps:** 型定義あり・依存ありのパッケージは **85個**（計671ファイル, 約6.3万行） [17]。
- **Never Typed, no deps:** 型定義なし・依存なしのパッケージは **102個**（計255ファイル, 約2.1万行） [17]。
- **Never Typed, with deps:** 型定義なし・依存ありのパッケージは **40個**（計544ファイル, 約1.9万行） [17]。

合計で513パッケージ・4,162ファイル・約22.6万行のコードからなる評価ベンチマークとなっています [17]。

3. 評価方法と指標

使用モデルと設定: TypeWeaverの評価には、研究文献から選ばれた3種類の型予測モデルが用いられました [19]。一つ目は**DeepTyper**（再帰型ニューラルネットワーク; RNN） [20]、二つ目は**LambdaNet**（プログラムをグラフ構造とみなしたグラフニューラルネット; GNN） [20]、三つ目は**InCoder**（コード補完に特化した汎用のTransformerモデル。特に途中部分の穴埋め「fill-in-the-middle」をサポートする大型言語モデル）です [20]。これらはいずれもJavaScript/TypeScriptの型推論に関して先行する手法であり、アーキテクチャが大きく異なる点を選定理由としています [12]。TypeWeaverは各モデルから得た型予測結果を入力として前述の型織り込みを行い、生成されたTypeScriptコードをTypeScriptコンパイラ（`tsc`）でコンパイルして評価しました。コンパイラは`--noEmit`（コード生成しない）等のフラグを立て、**型チェック**のみを行う設定です [21]（`--esModuleInterop`や`--skipLibCheck`を有効化し、Strictモードは無効に設定）。

成功判定の定義: 評価の主指標は、**型移行後のコードが型チェックを通過したか**どうかです。これをパッケージ単位とファイル単位の2段階で測定しています。

- パッケージ単位の成功率: あるパッケージ内の**全てのファイル**に対して型エラーが一つも出なければ、そのパッケージの移行は成功とみなされます。一箇所でも型エラーが残ればパッケージ全体としては失敗です [4]。非常に厳格な基準であり、「1つでも不正確な型注釈があればパッケージ全体が型検査エラーで失敗する」ことになります [4]。なお、モデルや変換工程によっては一部パッケージでツール自体がタイムアウト・エラーになる場合もありましたが、その場合それらパッケージは除外し、**TypeWeaverが最後まで処理できたパッケージの中で**何%が型検査成功したか、という比率で報告しています [22] [23]。

- ファイル単位の成功率: 上記が厳しい基準のため、**より細粒度な指標**として「各パッケージ内の何割のファイルがエラーなく型検査を通ったか」も測定しています [24] [25]。これはパッケージごとのエラーのないファイルの割合を算出し、全体傾向を見るものです。TypeScriptではファイルがモジュール単位となっており、ファイル内にエラーが無ければ「そのモジュール内部およびインポートしている型との整合性が取れている」ことを意味します [21]。したがって移行時には、一旦エラーの無いファイルは脇におき、エラーのあるファイルに開発者が注力するといった運用も可能になります [26]。この指標により、完全移行に至らないパッケージでも部分的な進捗を評価できます。

型注釈の質の評価: 型検査エラーの有無だけでなく、挿入された型注釈そのものの質も分析しています。

- トリビアルな型の割合: モデルが付与した型注釈の中に、`any`や`any[]`、あるいは`Function`型（引数・戻り値がすべてany扱いの関数型）といった**万能すぎて意味の薄い型**がどの程度含まれるかを測定しました [27]。これらの注釈は付ければ確かに型エラーは隠せますが、開発者に有用な型情報を与えないため「トリビアルな型注釈」として分類しています [27]。成功した（エラーのない）ファイル内に占めるトリビアル型の割合をモデル毎に集計し、型注釈の有用性を評価指標としました。

- 手書き型との一致精度（Accuracy): データセット内の「DefinitelyTyped」パッケージでは、人間が記述した正解の型定義（.d.tsファイル）が存在します。この**既存の手書き型**をグラウンドトゥルース（正解）とみなし、TypeWeaverが生成した型注釈とどの程度一致しているかを精度（Accuracy）として算出しました [28] [29]。具体的には、TypeScriptコンパイラに型チェック時の自動出力機能（`*.d.ts`宣言ファイルの生成）を利用させ、移行後コードから得た型宣言と手書きの型宣言を比較しています [30] [31]。関数シグネチャ（関数名・引数型・戻り値型）に着目し、両者に同名の関数が存在する場合にその**引数型リストと戻り値型が完全一致**しているかを確認します [31]。型の修飾子（例: readonly）は無視し、ユニオン型の順序が異なる場合（`string|number` vs `number|string`）

は文字列一致にはカウントしません [32] 。また文献に倣い、正解が `any` の場合は比較をスキップしています [33] 。この基準で一致率を算出することで、「モデル予測が人間の付けた型とどれだけ合致したか」を評価しました [34] 。

コンパイルエラーの分析: 型移行に伴い発生するTypeScriptコンパイルエラーの**種類と頻度**についても詳細に調査しています。TypeScriptコンパイラのエラーには番号が振られているため分類が容易であり [35] 、上位10種類のエラーコードを集計してその原因を解析しました [35] [36] 。代表的なエラーとして報告されているのは、以下のような**型不整合に関連するもの**です [37] :

- プロパティ不存在エラー (例: TS2339, TS2551) – あるオブジェクト型に存在しないプロパティにアクセスしている。
- 代入の型不一致 (TS2322) – ある変数に型の異なる値を代入している。
- 関数呼び出し時の引数型不一致 (TS2345) – 関数に渡した引数の型がパラメータの型と適合しない。
- 非関数型の呼び出し (TS2349) – 関数ではない値に関数呼び出しの括弧 `()` を付けてしまっている（予測型が間違って関数以外になっているケース）。
- 型の異なる値同士の比較 (TS2367) – 異なる型同士を条件分岐で比較している（常にfalseになるような比較）。
- ジェネリック型の型引数欠如 (TS2314) – ジェネリック型を使用しているが必要な型パラメータを提供していない [38] 。この原因として、DeepTyperやLambdaNetがジェネリック型を十分扱えず型引数予測が欠落することが挙げられています [38] 。

なお、これ以外にも型に直接起因しないエラーとして、**未定義識別子の参照** (TS2304: name not found) や **モジュールのインポートエラー** (TS2307: module not found) なども一部で発生しました。後者は特にCommonJSからESMへの変換時にモジュール名解決が合わず生じるケースがあり、モジュール形式変換の難しさを示唆しています。

4. 評価結果の要点

型チェック成功率（パッケージ／ファイル）: 結果として、**パッケージ単位で完全に型チェックを通過できたものはごく少数**でした。モデル中最も高かったものでも全体の約**21%のパッケージ**しかエラーなくコンパイルできず [39] 、残り約8割は何らかの型エラーが残る結果となりました。DeepTyperとInCoderはほぼ同程度の成功率で、**約20%前後**のパッケージが型検査に合格しましたが、**LambdaNetは約9%と低く**、他モデルとの差が顕著です [40] [41] 。特に外部依存を持つパッケージでは失敗が多く、依存なしのパッケージに比べ成功率が低い傾向が全モデルで見られます [42] 。一方、**ファイル単位**で見ると状況は改善します。モデル最高性能のInCoderでは全ファイルの約**69%**がエラーなしにコンパイル成功し [39] 、DeepTyperでも43.4%、LambdaNetでも25.2%のファイルは個別には問題なく型付けできています [43] 。つまり多くのパッケージでは「エラーのあるファイル」が一部に留まり、**大半のファイルは自動型付けでエラーゼロにできている**ことが示されました [44] 。この観点からは、**約7割のファイルは修正不要**で残りの3割のファイルに注力すればよいという見方ができ、完全移行に至らない場合でも部分的な有用性が確認できています [45] 。なおファイル単位の結果では、依存の有無による成功率差は統計的に明確ではなく、依存を含むパッケージでも個々のファイル成功率は依存なしパッケージと遜色ないことがわかりました [46] [47] 。

各モデルの特徴と精度‐成功率ギャップ: モデルごとの詳細を見ると、「型注釈予測の精度」と「型チェック成功率」のギャップが浮き彫りになっています。DeepTyperとInCoderは型検査通過率で優位でしたが、その手法として**多くの「any的」な安易な型注釈に頼っていた**ことがデータから分かります。例えばDeepTyperが生成した型注釈の約**60%**は `any/any[]/Function` といったトリビアル型で占められており、InCoderも約40%がトリビアルでした [27] 。一方LambdaNetはトリビアル型の割合が**約25%**と最も低く、より具体的な型推論を試みていることが伺えます [27] 。しかしその分、LambdaNetは無理に具体的な型を付けた結果として型不整合を多発させ、**型検査を通せるコードは少なくなってしまった**と言えます。実際、LambdaNetは手書き型との一致率（Accuracy）では他モデルを上回りました。全体精度は**約41.5%**で、DeepTyperの約32.1%や

InCoderの約27.3%より高い値を示しています [48]。この傾向は先行研究と同様で、**LambdaNetは型予測精度自体は高いが型チェック観点の成果は劣る**という逆転現象が確認されました [34]。DeepTyperは予測精度こそ低めですが、その保守的な出力（多数のany利用）のおかげで**LambdaNetの2倍以上のパッケージを型検査合格させる**ことに成功しています [41] [27]。InCoderは中庸的な戦略をとっており、精度・成功率ともに中間の値ですが、ファイル単位成功率では最高の結果を出しました [43]。これはInCoderが汎用大規模モデルとして多様な文脈に対応できたことを示唆します。総じて、モデル間で「安全だが役に立たない型を付けてでも通す」戦略（DeepTyper）と「攻めた型推論をして外す」戦略（LambdaNet）の対比が見られ、TypeWeaverを通じて**精度評価と型移行実効性のギャップ**が明確に浮かび上がった形です [34]。

モジュール形式変換（CommonJS→ESM）の影響: TypeWeaverが提供する**モジュール構文の自動変換**は、モデルの種類によって成功率に異なる影響を及ぼしました。著者らはCommonJSのままのコードとESMに変換したコードの両方で型移行を実行し、結果を比較しています。その結果、**パッケージ単位の成功率**については変換によって悪化するケースが見られました。DeepTyperでは変換なしの場合の成功率25.4%が、変換後には20.7%へ低下し、InCoderも21.3%から19.1%へと僅かながら下がりました [49]。LambdaNetのみ8.4%から8.9%へとわずかに改善しましたが、全体としてモジュール変換はパッケージ全体の合格率を高める決定打にはなっていません [49] [50]。一方で**ファイル単位の成功率**を見ると、モデルによって明暗が分かれます。LambdaNetは変換後に23.5%→25.2%と僅かに向上し、InCoderは**56.1%→69.2%と大幅に向上**しました [51]。特にInCoderに関しては、「型定義なし・依存あり (Never typed, with deps)」カテゴリのファイル成功率が**11.2%から88.6%へ跳ね上がる**という顕著な改善が観測されています [51]。これはESM変換により外部ライブラリの型情報がより正確に伝搬し、InCoderの生成した型注釈と噛み合った結果と推測されます。一方でDeepTyperは変換によって48.1%→43.4%へと成功率が下がり、特に「型定義なし・依存あり」パッケージでは64.5%から34.7%へ**大幅悪化**しています [51]。DeepTyperの場合、変換前は `require` 経由で入ってくる型を包括的に `any` 扱いすることでエラーを隠せていたものが、変換後に詳細な型情報が露呈した結果エラーが増えた可能性があります。以上より、**モジュール形式の変換は一長一短**であり、モデルとコードパターン次第で型検査成功率を高めることにも下げることにもなり得ることが分かりました。

総括: 本研究の成果は、機械学習による型予測モデルを用いたTypeScript自動移行の現状と課題を浮き彫りにしています。**完全自動での型移行成功**は容易ではないものの、ファイル粒度で見れば多くのコード片が自動で型付け可能であり、移行作業の労力削減に資することが示されました [44]。一方で、モデルの出力する型の**有用性（精度）**と**実用上の型検査合格率**にはトレードオフがあることが確認され、評価指標として型検査パス率を導入する意義が強調されています [2] [34]。著者らは「**型検査に通した後**」のプログラム挙動（ランタイムエラーがないか等）まで評価範囲を広げるべき可能性にも言及しており [52]、今後のTypeScript自動型付けツールの改良や評価基準の発展につながる知見を提供しています。

参考文献: TypeWeaverに関する詳しい内容は Ming-Ho Yee らによるECOOP 2023論文「Do Machine Learning Models Produce TypeScript Types That Type Check?」 [53] および講演スライド [54] [19] にまとめられています。本レポートでは同論文の要点をまとめました。各種データは論文中の表や図（Table 1〜8, Figure 5〜12）から引用しています。

---

[1] [19] [20] [54] Do Machine Learning Models Produce TypeScript Types that Type Check? (ECOOP 2023 - Research Papers) - ECOOP 2023
https://2023.ecoop.org/details/ecoop-2023-papers/11/Do-Machine-Learning-Models-Produce-TypeScript-Types-that-Type-Check-

[2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [21] [22] [23] [24] [25] [26] [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] 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