BERT系モデルとコンパイラフィードバックによるJavaScriptライブラリ向け TypeScript宣言ファイル（.d.ts）自動合成

論文形式サンプル（概要把握用） / 2025-12-17 作成

※本PDFは「提案手法で想定通りの結果が出た」場合の論文の流れを掴むためのドラフトです。数値はすべて例（仮）であり、実験結果に置換してください。

要旨

TypeScript（TS）プロジェクトにおいて、型定義（.d.ts）が不十分または欠如したJavaScript（JS）ライブラリの利用は、型安全性と開発体験を著しく損なう。
従来の学習ベース型推論研究は、個々の注釈の正解率（Top-1/Top-k accuracy）を中心に評価する傾向があるが、実運用では「下流のTSプロジェクトがコンパイル（type check）を通るか」という移行成功が決定的である。 本研究は、公開API境界（exportされたシンボル）に対する宣言ファイル（.d.ts）合成を対象とし、BERT系のマスク予測型モデル（TypeBERT型）で得たTop-k候補を、TypeScriptコンパイラ（tsc）のエラー情報で誘導しながら反復的に修正する手法を提案する。
JSライブラリ群とその下流TSクライアント群を用いた評価（例）において、提案法は all-any / all-unknown / Top-1のみのベースラインを上回る package-level type check 率を達成し、過度な any による“ トリビアル成功 ”を抑えつつ改善できることを示す。 さらに、探索（Top-k切替）とフォールバック規則（unknown/union化等）の寄与をアブレーションで分析し、残る失敗要因（クロスファイル整合性、ジェネリクス/オーバーロード、依存先型不足）を体系化する。

1. はじめに

TypeScriptはJavaScriptのスーパーセットとして広く普及している一方、エコシステムには型定義が不足するJSライブラリが数多く存在する。 その結果、利用側は any による型安全性の喪失、または型不整合によるコンパイル障害に直面する。 学習ベースの型推論は有望だが、重要なのは「それっぽい型を当てる」ことよりも「型を統合した結果として、プロジェクトが型チェックに通る」ことである。 Yee and GuhaはTypeWeaverを通じて、精度が高いと報告されるモデルでも、パッケージ単位で type check が通る割合が限定的であることを示した（例：最良モデルでも約21%のパッケージがtype check）[1]。
また、LLMを用いるOpenTauも、単純生成ではなく探索・分解・コンパイラによる候補評価を導入して type check を最大化する枠組みを採用している[2]。
これらは「局所精度中心の評価」から「移行成功中心の評価」へ軸足を移す必要性を示唆する。 本研究は、BERT系の再現性の高い枠組み（TypeBERT）[3]を基盤に、公開API境界の.d.ts合成をコンパイラ誘導で最適化し、下流TSプロジェクトに対する移行成功（type check）を改善することを目的とする。

2. 背景と動機

2.1 局所精度と移行成功のギャップ 学習モデルが個々の変数や関数に対して正しそうな型を付与できても、別ファイルの呼び出し、同一識別子の多用途利用、依存ライブラリ型不足の連鎖などにより、プロジェクト全体で整合性が崩れる。 TypeWeaverはこの点を実証的に扱い、(i) 依存型取り込み、(ii) モジュール変換、(iii) 型の織り込み（weaving）、(iv) 非型予測の除去など、モデル外の工程が移行の成否を左右することを示した[1]。

2.2 LLM系手法との位置づけ
OpenTauは、LLMの生成結果を候補として保持し、TypeScriptコンパイラで候補を評価し、エラー数とtypedness を基にランキングする[2]。 これは、生成モデルの能力だけでなく、探索（search）と型チェックフィードバックが移行成功に不可欠であることを示す。

2.3 BERT系（TypeBERT）の利点 TypeBERTは大規模データに基づき、コードをトークン列として扱うBERT系モデルで型注釈を予測し、SOTAに競争力のある精度を報告した[3]。
一方で、TypeWeaverが示す通り、精度が高くても移行成功とは限らないため、本研究は TypeBERT を“ 型予測器 ”ではなく“ 移行を成立させる部品 ”として再設計する。

3. 問題設定

入力：
(1) JSライブラリ L のソースコードと公開API（exportされたシンボル集合 S）
(2) L を依存に持つ下流TSプロジェクト集合 C(L)（評価用）
(3) （任意）下流側での利用情報（呼び出し例、戻り値の利用方法など）

出力：
L の公開APIに対する TypeScript 宣言ファイル D(L)（例：index.d.ts）。

目的：
下流プロジェクトが tsc --noEmit で型チェックに通る割合（package-level type check）を最大化しつつ、any等のトリビアル型付与を抑制する。

主要指標：
・TypeCheckRate：C(L) のうち type check に成功した割合
・ErrorCount：コンパイラエラー数（失敗時も改善量を測る）
・TrivialRate：any/Function 等の割合（定義は実験節で固定）

4. 提案手法

本手法は「候補生成」と「コンパイラ誘導の反復修正」の2段階からなる。
まず、BERT系モデルで公開API各要素の型をTop-k候補として生成し、初期の.d.tsを合成する。
次に、下流TSプロジェクトに対して tsc を実行し、得られたエラー情報に基づき、関係するシンボルのみ候補を差し替える（または安全側にフォールバックする）ことで、type check の成立率を押し上げる。

4.1 公開API抽出

JSライブラリのエントリポイントおよび再輸出を解析し、公開API（exportされた関数・クラス・定数・オブジェクトメンバ）を抽出する。 CommonJS（module.exports / exports.x）とESM（export / export default）の双方を対象とする。
抽出結果はシンボル単位の予測対象として正規化し、.d.ts生成の骨格とする。

4.2 Top-k型候補生成（TypeBERT型）

各シンボル s ∈ S に対し、宣言付近のコードコンテキスト（一定トークン窓）を入力として、マスク予測により型候補を生成する。 出力は Top-k（例：k=10）候補の列とする。 フェーズ1では型語彙を制約し、基本型・配列・Promise・単純なunion・unknown・any等に限定する（拡張は将来課題）。

4.3 .d.ts合成（weaving）

Top-1候補を用いて初期宣言ファイル D0(L) を合成する。 例：exportされる関数 f に対し「declare function f(a: T1, b: T2): R;」を生成し、モジュール宣言として整形する。
この段階で“ 型として構文的に不正 ”な候補（非型トークン列）は除去または置換する。

4.4 コンパイラ誘導の反復修正（提案の中核）

下流TSプロジェクト群に対して tsc --noEmit を実行し、エラーを収集する。 エラーが発生した場合、影響の大きいシンボル（当該APIや型が参照される箇所）に限定して候補を差し替え、再度型チェックを行う。 全体を再予測しないことで探索空間を抑え、実用的な計算コストで成立率を改善する。

```
Algorithm 1: Compiler-Guided Candidate Switching (CGCS)
Input: candidates K(s) for each public symbol s, initial selection sel(s)=1
for iter = 1..N do
  run tsc --noEmit on downstream projects with current .d.ts
  if no errors then break (success)
  identify implicated symbols S_E from compiler errors
  for s in priority(S_E) do
    try sel(s) <- sel(s)+1 (switch to next candidate)
    re-run tsc; keep switch if error count decreases else revert
  end for
  if no improvement then apply fallback rules (unknown/union/any)
end for
Output: synthesized .d.ts
```

4.5 フォールバック規則（移行成功を優先）

候補切替で改善しない場合、次の順で安全側に寄せる。 (1)
引数型：unknown（操作が必要な場合に限りany） (2) 競合する呼び出しがある場合：union化（例：string | number） (3) 戻り値型：unknown（メソッドチェーン破綻が顕著な場合はany）“ anyに逃げれば通る ”だけの結果を避けるため、TrivialRateを測定し、成立率とのトレードオフとして報告する。

5. 実験設定

5.1 データセット構築（例）
・JSライブラリ：npm上位から、型定義が無い/弱いパッケージを抽出（例：200件）。
・下流TSクライアント：各ライブラリを依存に持つGitHubプロジェクトを収集（例：合計1,000件）。
・補助的な正解（任意）：@types（DefinitelyTyped）または過去のTS化コミットを用意。

5.2 ベースライン
BL1: all-any（公開APIをすべて any）
BL2: all-unknown（公開APIをすべて unknown）
BL3: Top-1のみ（TypeBERT Top-1を挿入、反復修正なし）
BL4: Top-k（モデルスコアのみで選択、tsc誘導なし）

5.3 実行環境
TypeScriptのバージョン、tsconfig、依存解決をDocker等で固定し、再現可能な評価を行う。
主要評価は tsc --noEmit の成否とエラー数である。

6. 結果（例：数値はダミー）

※以下の表は説明用の仮数値です。実験で得た実測値に置換してください。

手法	Package-level type check ↑	平均エラー数 ↓	Trivial率(any) ↓
BL1: all-any	62%	120	95%
BL2: all-unknown	28%	310	0%
BL3: Top-1のみ	41%	210	18%
BL4: Top-k（tsc誘導なし）	45%	195	18%
提案: Top-k + tsc誘導反復修正	57%	140	12%

観察（例）：
・Top-1のみでは局所的に妥当な型でも下流で整合性が崩れ、成立率が伸びにくい。
・Top-k候補を tsc のエラーで誘導して切替・フォールバックすることで、成立率が大きく改善する。
・all-anyは成立率が高いが、型情報としてはトリビアルである。提案法は any
依存を抑えつつ成立率を引き上げる点に価値がある。

7. アブレーション（例）

設定	Package-level type check ↑	メモ
提案法（フル）	57%	Top-k + tsc誘導 + フォールバック
tsc誘導なし	45%	Top-kを活かしきれない
フォールバックなし	49%	収束しない/詰まるケースが増える
利用情報なし	53%	呼び出し側情報が成立率に寄与（例）

探索（候補切替）とフォールバック規則が成立率の改善に寄与することが確認できる（例）。

8. 失敗分析（例）

失敗ケースを分類すると、次の要因が支配的であることが多い。
(1) クロスファイル整合性：同一シンボルが別ファイルで異なる用途で利用され、単一型では矛盾する。
(2) ジェネリクス/オーバーロード：閉じた語彙の単純型では表現できないAPI形状がある。
(3) 依存先型不足の連鎖：依存ライブラリの型定義欠如がエラーとして伝播する。
(4) 高度に動的なJS：実行時に形が変わるオブジェクト等、静的宣言化が困難。
これらは「局所精度が高い」だけでは解決しにくく、移行成功を評価する枠組みの重要性を裏付ける[1]。

9. 関連研究

TypeBERTは大規模事前学習＋微調整により、トークン列ベースのBERTモデルで型注釈予測を行う枠組みを示した[3]。
DiverseTyperはユーザ定義型（クラス/インタフェース等）の長い裾野分布に対応するための学習戦略を導入した[4]。
TypeWeaverは、型予測モデルを実際にコードへ統合して type check
を評価する移行ツールおよびデータセットを提示し、局所精度中心評価の限界を示した[1]。
OpenTauはLLM候補生成に探索・分解・コンパイラ評価を組み合わせ、type check の改善を報告した[2]。
さらに、TypeScript宣言ファイルの推論・進化を支援する研究として Kristensen and Møller の

tsinfer/tsevolve がある[5]。
本研究は、(i) 公開API境界の.d.ts合成に焦点を当て、(ii) BERT系Top-k候補を tsc
フィードバックで反復修正して、(iii) 移行成功（type
check）を主要目的関数として最適化する点で差別化される。

10. 限界と今後の課題

本稿の手法は、探索空間を抑えるために型語彙を制約し、公開API境界に対象を絞っている。
そのため、複雑なジェネリクス、条件型、オーバーロード、依存型推論などは今後の拡張課題である。
また、下流プロジェクト収集の偏りや、TypeScriptバージョン差、tsconfig差による外的妥当性のリスクがある。
将来は、(i) 型語彙拡張（ユーザ定義型の参照・生成）、(ii) 依存先型の同時推論、(iii)
失敗箇所のより精密な最小修正（MUS的）探索、(iv)
クローズドソース環境を想定した“ 利用側観測中心 ”の推論へ拡張する。

11. 結論

本稿は、JSライブラリの公開APIに対するTypeScript宣言ファイル（.d.ts）合成において、BERT系モデルのTop-k候補と
TypeScriptコンパイラ（tsc）フィードバックを組み合わせた反復修正により、移行成功（type
check）を最適化する枠組みを提示した。 局所精度では捉えにくい“ 統合の難しさ ”を主要目的関数として扱うことで、実運用に近い評価・改善が可能になる。

参考文献

[1] Ming-Ho Yee, Arjun Guha.“ Do Machine Learning Models Produce TypeScript Types That Type Check? ” ECOOP 2023 (LIPIcs), 2023. arXiv:2302.12163.

[2] Federico Cassano et al.“ Type Prediction With Program Decomposition and Fill-in-the-Type Training. ” arXiv:2305.17145, 2023. (OpenTau).

[3] Kevin Jesse, Premkumar T. Devanbu, Toufique Ahmed.“ Learning Type Annotation: Is Big Data Enough? ” Proceedings of ESEC/FSE 2021. DOI: 10.1145/3468264.3473135.

[4] Kevin Jesse, Premkumar Devanbu, Anand Ashok Sawant.“ Learning to Predict User-Defined Types. ” IEEE Transactions on Software Engineering, 2022. DOI: 10.1109/TSE.2022.3178945.

[5] Erik Krogh Kristensen, Anders Møller.“ Inference and Evolution of TypeScript Declaration Files. ” FASE 2017. DOI: 10.1007/978-3-662-54494-5_6.