

大規模言語モデルを用いた逆コンパイルに おけるエラー原因の体系的な調査

オープンソースソフトウェア工学研究室

学生番号 : 60276216

発表者: 福井 遥陽

背景

逆コンパイルとは

低水準言語のコードから高水準言語(C,Python,java等)のコードを得る手段.

```
arduino
b8 04 00 00 00 ; mov  eax, 4      ; sys_write
bb 01 00 00 00 ; mov  ebx, 1      ; stdout
b9 0a 10 40 00 ; mov  ecx, 0x40100a ; 文字列のアドレス
ba 0d 00 00 00 ; mov  edx, 13      ; 文字列長
cd 80          ; int   0x80      ; システムコール
```

アセンブリコード

逆コンパイル



```
int add(int a, int b) {
    return a + b;
}
```

逆コンパイル後
ソースコード

逆コンパイルの目的

- ・ マルウェアの解析
- ・ ソフトウェアの互換性の維持等



元のソースコードが手に入らない状況

高水準のコードからプログラム全体の処理を追うことが可能となる

既存ツール

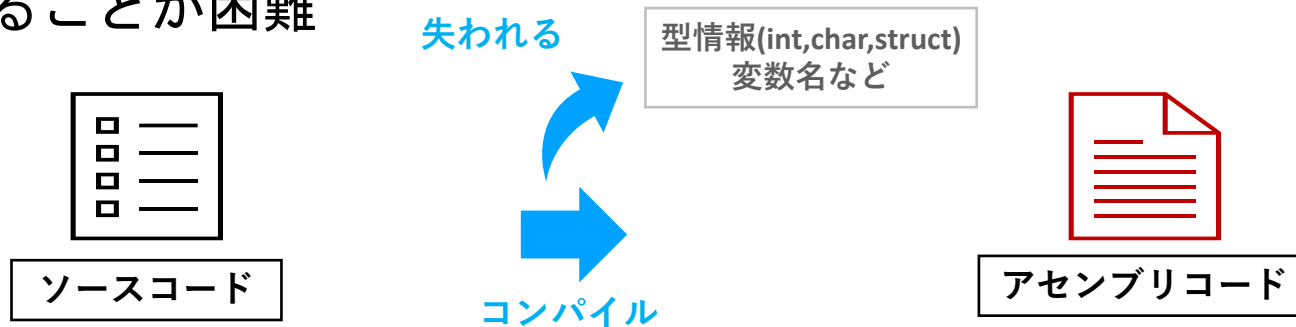
現状の逆コンパイルツール

例) Ghidra^[1] (制御フロー解析等に基づくルールベースの逆コンパイルツール)

既存ツールの課題

ルールベースであり, 既にコンパイルの段階で失われた意味情報を復元することが困難

例)



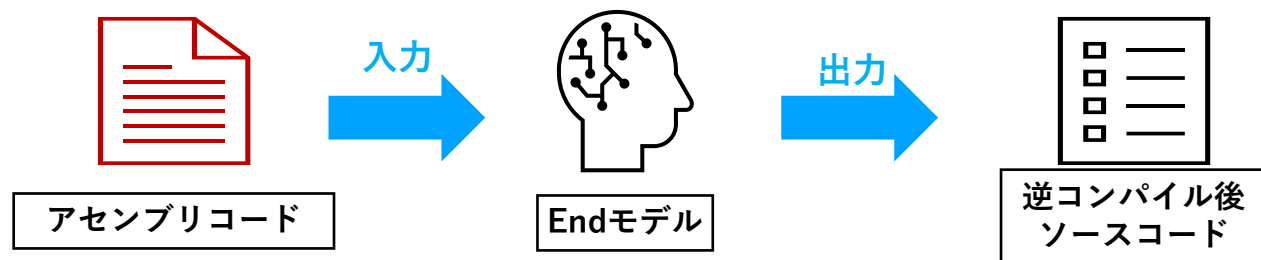
→大規模言語モデル(LLM)で意味情報の補完を試みる手法に遷移

[1] Hanzhuo Tan, Qi Luo, Jing Li, Yuqun Zhang “LLM4Decompile: Decompling Binary Code with Large Language Models”

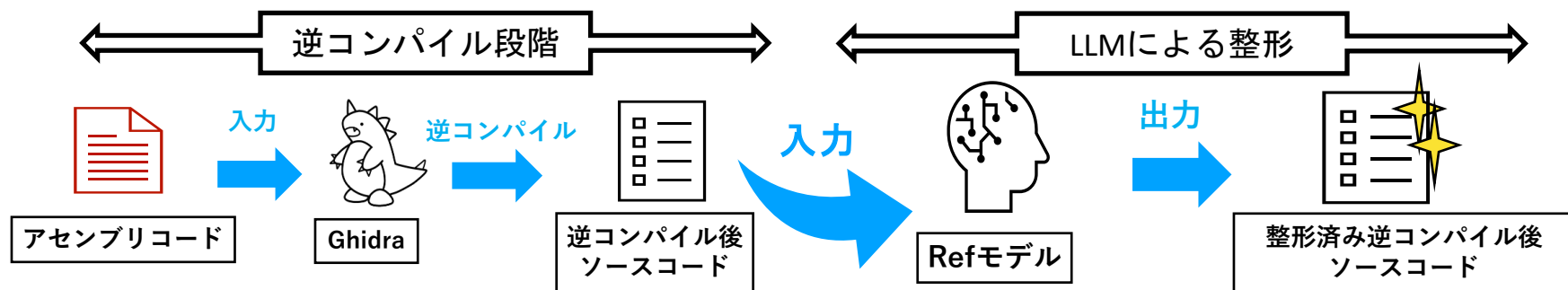
先行研究

逆コンパイル特化LLMモデル

- LLM4Decompile-End (Endモデル) [1]



- LLM4Decompile-Ref (Refモデル) [1]



先行研究

貢献

End/Refモデルは共に既存モデル（ChatGPT-4o等）より高い再実行可能性を示した。^[1]

・再実行可能性：コンパイル・実行可能且つ、自動テストを通過する精度を測る指標

課題

実行不可能であるコードの内36%を超えるコードがコンパイル不能であり、コンパイルエラー原因が不明^[1]

原因を特定・改善することで再実行可能性の向上を期待

目的とアプローチ

研究目的

逆コンパイル特化LLMの起こすコンパイルエラー原因を体系的に明らかにする

アプローチ

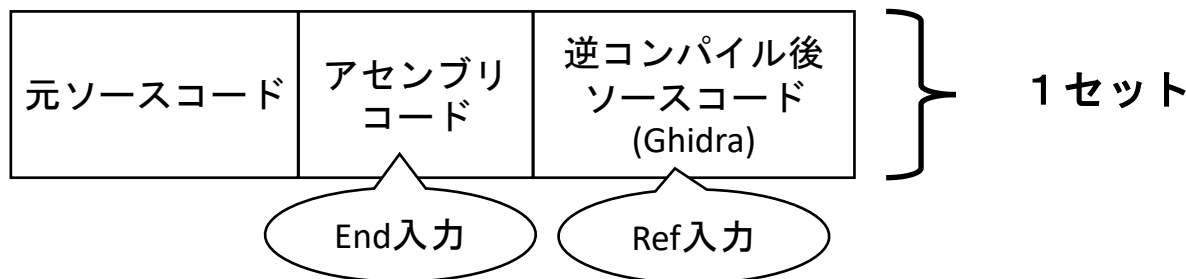
- End / Refモデルの生成コードを対象
- gcc（Cコンパイラ）で検出されたコンパイルエラーをカテゴリ別に判定・集計
- データセット別にエラー傾向を比較（モデル依存かデータ依存かの検証）

実験設定

使用データセット

- humanEval-decompile(Humaneval)とMBPP-decompile(MBPP)を使用^[2]
→ 先行研究の著者がhumanevalとMBPPをC言語用に変換したデータ
※ Humanevalデータの全164件, MBPPデータの全1948件中400件を使用

表：データセットの構成



使用逆コンパイルモデル

- LLM4Decompile-End(6.7B) ^[3]
- LLM4Decompile-Ref(6.7B) ^[4]

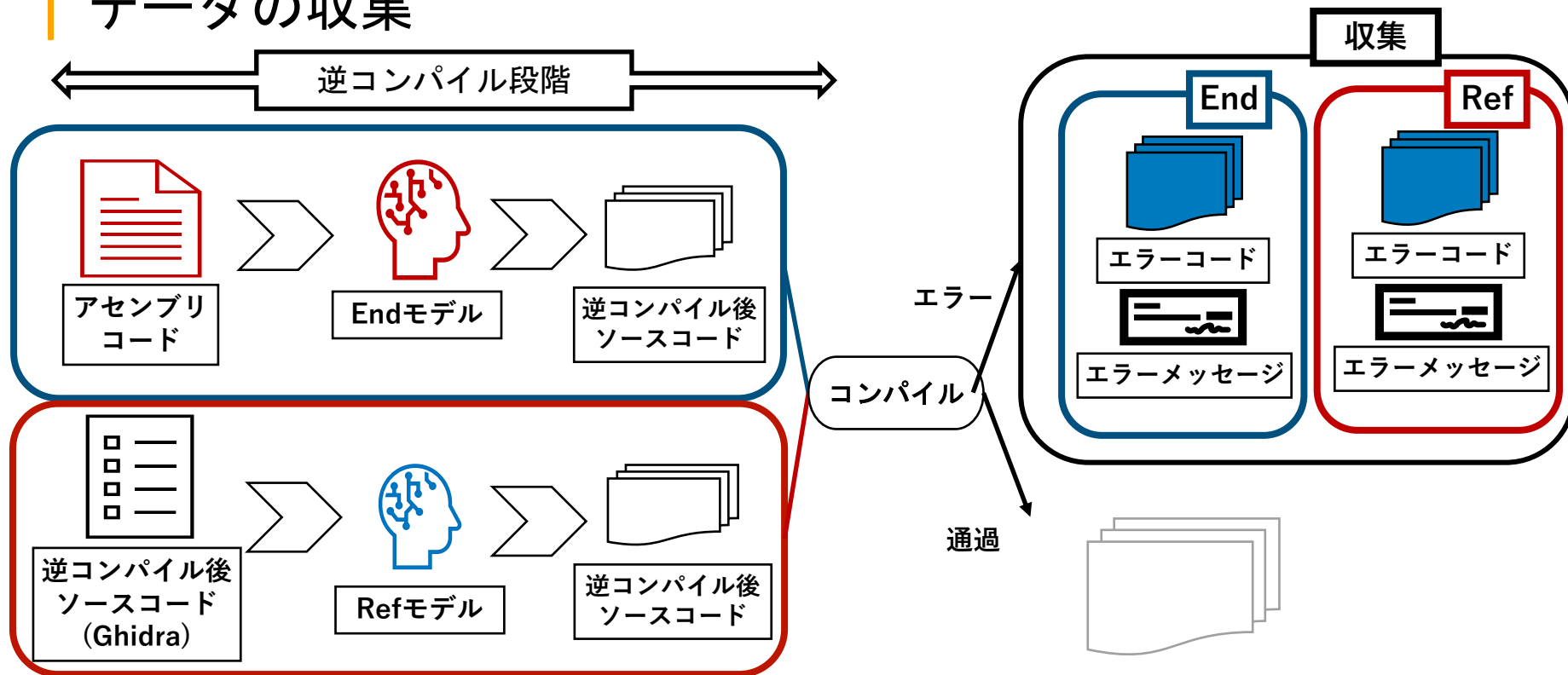
[2] Tan et al., *LLM4Decompile decompile-bench* (HumanEval-decompile, MBPP-decompile), GitHub, 2024

[3] Tan et al., *LLM4Binary/llm4decompile-6.7b-v1.5*, Hugging Face Model Hub (2024)

[4] Tan et al., *LLM4Binary/llm4decompile-6.7b-v2*, Hugging Face Model Hub (2024)

実験設定

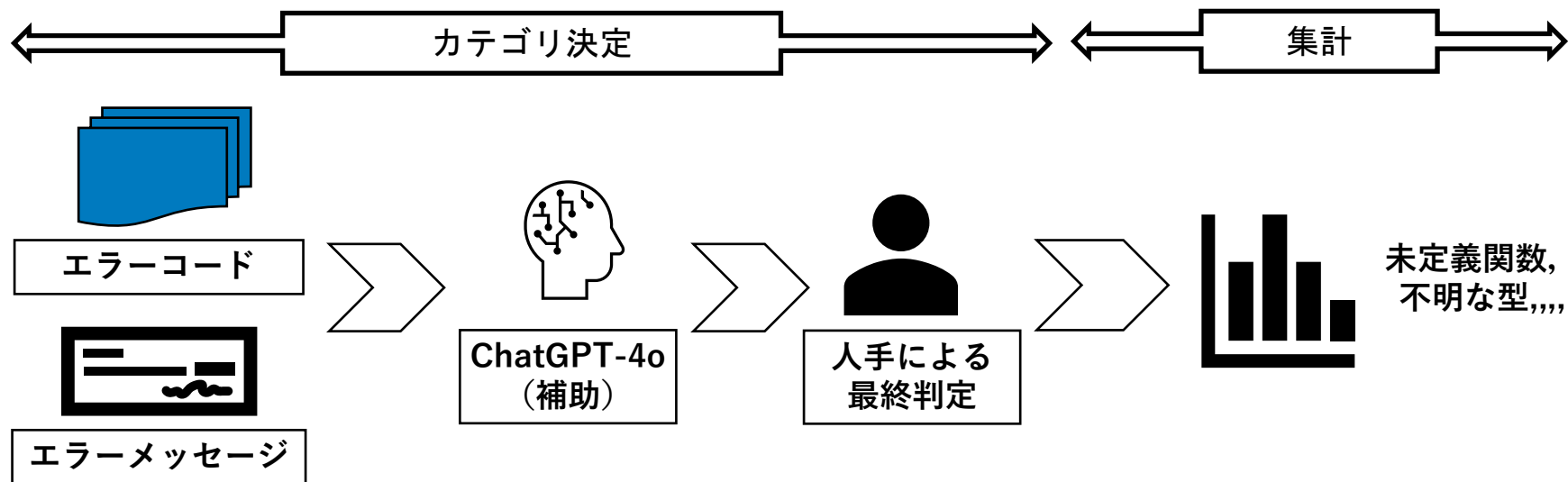
データの収集



- End/Refモデル出力コードを別々にコンパイル
- gcc (-std=c99, -fsyntax-only) で構文エラーのみ検出
- コンパイルエラー発生コードを対象

実験設定

結果判定・集計



入力：エラーコード，エラーメッセージ

補助：ChatGPT-4o（判定支援）

最終判定：人手による確認

※End/Humanevalで104件，Ref/Humanevalで60件

End/MBPPで110件，Ref/MBPPで74件のエラーコード収集

実験設定

表：エラー原因カテゴリ（一部抜粋）

カテゴリ	判別基準
未定義関数	定義されていない関数の呼び出し
未定義構造体	定義されていない構造体の呼び出し
不明な型の使用	C99で未定義の型の出現
型の不一致	異なる型間での代入・演算
ロジックの崩壊	処理が途中で終了しているコード

※エラーメッセージ準拠（可能な限り主観を排除）

実験結果 RQ1:モデル毎のエラー傾向

表：上位5件のエラー原因（Endモデル/Humaneval）

エラー原因	割合(%)
未定義関数	26.65
未定義構造体（引数）	25.00
ロジックの崩壊	11.54
不明な型（引数）	9.89
未定義構造体（関数内部）	6.04

未定義の識別子に関するエラーが頻出

表：上位5件のエラー原因（Refモデル/Humaneval）

エラー原因	割合(%)
不明な型の使用（関数内部）	20.00
型の不一致	19.17
不明な型の使用（戻り値）	15.83
不明な型の使用（引数）	15.00
ロジックの崩壊	10.83

型情報に関するエラーが頻出

実験結果 RQ2:データ間のエラー傾向の一貫性(補助的分析)

両データセット毎のエラー傾向（End/Refモデル）比較（主要なエラーを抜粋）

エラー原因	1.Humaneval差分 (End-Ref)(%)	2.MBPP差分 (End-Ref)(%)
未定義関数	+25.82	+20.62
未定義構造体（引数）	+22.50	+25.88
未定義構造体（関数内部）	+6.04	-5.11
ロジックの崩壊	+0.71	+12.50
不明な型（関数内部）	-20.00	-15.50
不明な型（戻り値）	-15.83	-11.06
不明な型（引数）	-5.11	-21.49
型の不一致（関数内部）	-17.24	-2.89

エラー傾向の一貫性あり

※正の傾向が強い：Endモデルで頻出
負の傾向が強い：Refモデルで頻出

・ 両データセットで同じ傾向が見られたため、モデル依存と考えられる

考察

Endモデル

未定義識別子のエラー：合計57.69%

→関数宣言部，構造体宣言部の復元が不十分？

func1()関数宣言無し

```
int func0(char *a, char *b) {  
    int i, j, k = 0;  
    i = strlen(a);  
    j = strlen(b);  
    for (i = 0; i < j; i++) {  
        if (!func1(a + i, b, i))  
            k++;  
    }  
    return k;  
}
```

未定義関数のエラー

改善策

- ・ 宣言部と関数本体の生成順序を明示する
プロンプト設計
- ・ 未定義識別子を自己検証させるプロンプト
の導入

考察

Refモデル

不明な型エラー：60.83%

型の不一致エラー：19.17%

→Ghidraの出力

に見られる表現（例：int_16,int32）

を多数確認

```
int8 func0(int32 *arr, int32 len) {  
    int32 i, j, k;  
    for (i = 0; i < len; i++) {  
        for (j = i + 1; j < len; j++) {  
            for (k = j + 1; k < len; k++) {  
                if (arr[i] + arr[j] + arr[k] == 0) {  
                    return 1;  
                }  
            }  
        }  
    }  
    return 0;  
}
```

不明な型のエラー

改善策

- ・ 出力コードの反復生成による型表記の正規化
- ・ ghidra表現の型を標準Cの型へ変換する指示をプロンプトに導入

まとめ

本研究の貢献

不明であった両モデルのコンパイルエラー原因を明らかにした.
→Endモデルでは**未定義識別子**, Refモデルでは**型情報の欠落**が主因

プロンプト設計及び学習方法の改善により再実行可能性の向上が期待される.

今後の展望

- ・ プロンプト設計
- ・ 未検証のデータセットを使用したエラー原因調査

院研究

背景

- 近年のデコンパイル研究では静的解析（制御フローグラフなど）を用いたプロンプト補助の研究が行われている

既存研究の問題

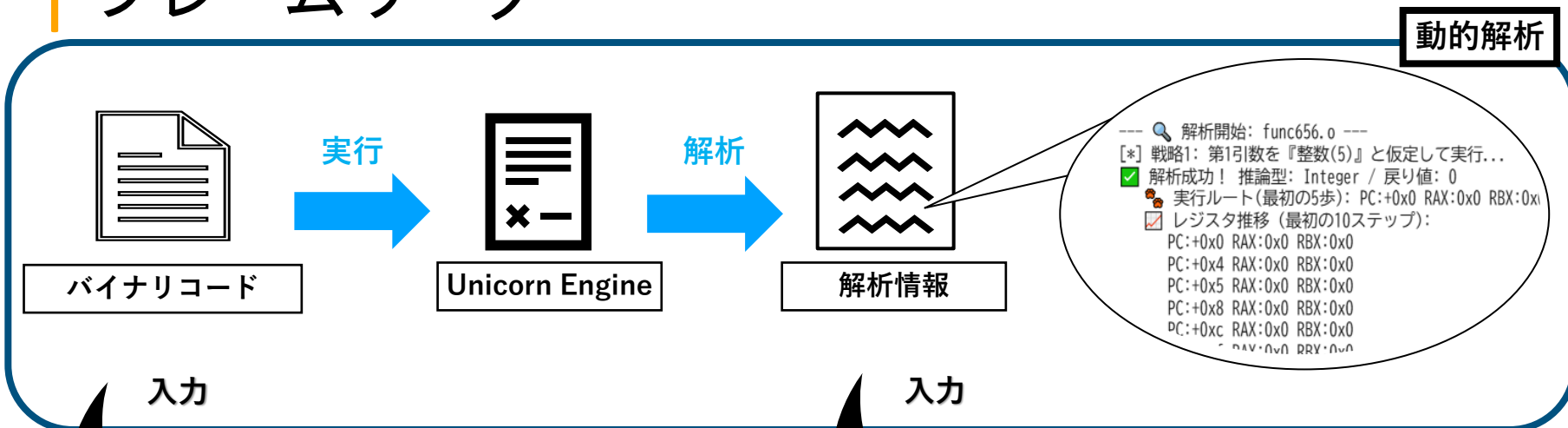
- 制御フロー解析などから得られる補助情報では復元不可能なものが多い。特に、バイナリから読み取れない情報（変数型など）は復元不可

目的

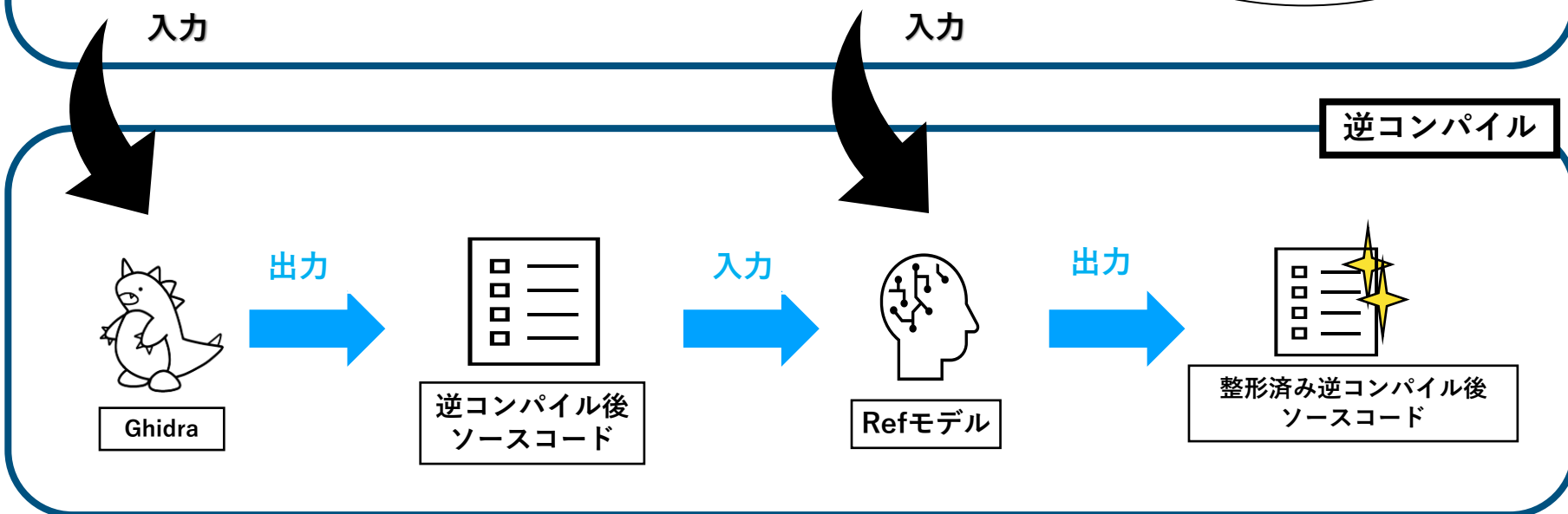
- 動的解析ツールを用いて間接的に失われた変数型を推測する情報を得る. それをプロンプトに導入
→既存手法に対して, LLM4Decompileモデルの再実行可能性の向上を期待する

フレームワーク

動的解析



逆コンパイル



院研究

使用データモデル

- LLM4Decompile-Ref(6.7B)

使用データセット

- Humanevalデータセット
- MBPPデータセット

院研究

| 実験結果 解析途中