東京大学
情報理工学系研究科 創造情報学専攻
修士論文

静的型付き言語における動的型生成に起因するエラーメッセージの改善

Improving error messages caused by dynamic type generation in statically typed languages

横井 駿平
Shumpei Yokoi

指導教員　千葉 滋 教授

2022 年 1 月

概要

リフレクション計算の一種である動的な型生成は，不正な型の生成による実行時エラーの発生時，その原因となる操作の場所がエラーの発生場所と離れているためエラー原因の特定が容易でない．本研究では，エラーの原因となりうる操作の実行時に操作内容とそのソースコード内における位置を記録し，エラー発生時に操作履歴から推定したエラー原因の位置を含んだエラーメッセージを生成する仕組みを提案する．静的な型エラーを含むプログラムをコンパイルすると，コンパイラはその原因に応じたエラーメッセージを生成する．これに習い，本研究では動的な型エラーをその原因によって分類し，類似するコンパイルエラーと対応付けた．そして，コンパイルエラーメッセージに類似した内容の実行時エラーメッセージを生成することが可能な動的型生成ライブラリを開発した．このライブラリは，操作履歴の情報を生成対象の型にメタデータとして埋め込み，型生成およびキャスト演算の実行時にメタデータを参照してエラーの原因を分析する．

Abstract

In dynamic type generation, which is a kind of reflective computation, when a runtime error is raised due to the generation of an invalid type, it is not easy to identify the cause of the error because the location of the operation that causes the error is far from the location where the error raised. In this study, we propose a mechanism that records the operation and its location in the source code at the time of execution of the operation that may cause the error, and generates error messages including the location of the error cause estimated from the operation history when the error is raised. When a program containing a static type error is compiled, the compiler generates error messages according to the cause of the error. In this study, we classified runtime type errors according to their causes and mapped them to related compilation errors. Then, we developed a dynamic type generation library that can generate runtime error messages similar to the compilation error messages. This library embeds operation history information as metadata in the types to be generated, and analyzes the causes of errors by referring to the metadata when genering types and executing cast operation.

目次

第 1 章

はじめに

C#はリフレクション計算の機能を備えた言語であり，標準ライブラリとして提供されている API を用いてコンパイル時には存在しない型を実行時に生成し利用することが可能である．実行時に型を生成する操作が必要な場面は通常のプログラムにはあまりないことからこの機能が使われるケースは限られているが，柔軟なプログラムを高速に動作させるため，リフレクションを応用した処理は主に高機能なライブラリの開発者によって用いられている．

この技術を用いたライブラリの例として，C#による Web アプリケーション開発用フレームワークである ASP.NET が挙げられる．Web アプリケーションにはその性質上，アプリケーション実行時に呼び出すメソッドを動的に決定する必要のある場面が存在する．リフレクションによってメソッドをその都度検索し呼び出すことでこの処理は実現可能だが，リフレクションによる動的なメソッド検索は通常のメソッド呼び出しの速度と比較して極めて低速であるため，実用アプリケーションで用いられるフレームワークの内部処理には適さない．そこで ASP.NET ではリフレクション操作によってメソッドを検索したのち，そのメソッドを直接呼び出すコードを動的に生成しキャッシュすることで 2 回目以降のメソッド呼び出しの高速化を実現している．

このように高機能なライブラリやフレームワークを作る上で重要になる動的型生成だが，リフレクション操作を行うプログラムは一般的なプログラム以上にバグを含みやすく，またエラーが生じた場合にその原因をエラーメッセージから推測することが難しいため，積極的に利用される機会はあまりなかった．リフレクション操作がバグを含みやすい事実は一般によく知られており，無用なバグを回避するためにリフレクションの使用はあまり推奨されていないことから，リフレクションを多用したプログラミングを簡単化するための研究はほとんど行われていない．本研究では動的型生成に起因するエラーメッセージを改善することで動的型生成を含むライブラリの開発における効率的なデバッグを実現する．

C#では，C#5.0 および C#10.0 においてメソッドの呼び出し元情報を取得する機能が追加された．この機能は主にデバッグ診断のためのもので，メソッドの呼び出しをログファイルに記録する場合や独自アサーションがエラーを検出した位置を報告する場合などに利用されている．この機能を発展させることで，実行時エラーが生じた際にそのエラーの発生位置だけではなくエラー原因の位置をエラーメッセージに含めることが可能になる．

本研究では，エラー原因の位置をエラーメッセージに含めることによる開発効率向上の実証研究として，C#における動的型生成 API にロギング機能を追加し，生成した型がエラーを発生させた際に型構築時の不備を指摘するための機構を提案する．動的に生成された型がエラーを発生させた場合，そのエラーメッセージはエラー原因の処理そのもののみを指摘する実行時例外に付随するものとなるため，エラーメッセージからバグの原因を特定するのは容易ではない．加えて，C#やその他の.NET 仮想マシン上で動作する言語ではコンパイルエラーとなるため記述することのできない不正な型やコードを生成することも出来てしまうために，一般的なプログラムでは目にすることのない実行時例外が発生する場合もある．この種の実行時エラーは静的なコードにおいてはコンパイル時に検出されるものであり，動的型生成に起因する動的エラーと型の不備に起因する静的エラーには関連性が見られる．

そこで本研究では，動的型生成 API の呼び出し履歴を対象の型にメタデータとして埋め込み，動的エラーが発生した際に埋め込まれたメタデータからエラー原因となった操作を指摘する拡張動的型生成 API を開発し，エラーメッセージ内にエラー原因の位置および関連するコンパイル時型エラーの ID を含めることでライブラリの開発効率を向上させた．また，実験の結果，動的に生成された型が型エラーを含んでいた場合原因の分析によってコンパイルエラーメッセージに近い内容の実行時エラーメッセージが生成出来ることを確認した．

以下，第 2 章では研究背景としてメタプログラミング，特にリフレクションについて紹介し，その扱いづらさを軽減するために行われてきた研究，そしてリフレクションが抱える問題について説明する．第 3 章では提案する拡張動的型生成 API の概要とその実装について説明する．第 4 章では本研究で開発した API を用いることでエラー原因の特定が容易になることを示す．最後に，第 5 章では本研究のまとめと今後の課題について述べる．

第 2 章

メタプログラミングとリフレクション

動的型生成は言語の提供するリフレクション API を介して行われ，生成された型は実行時環境に組み込まれる．型やコードを操作する技術には実行時リフレクションのように実行時に作用するものとテンプレートのようにコンパイル時に作用するものがあるが，これらはいずれもメタプログラミングの一種である．多くのメタプログラミング技法は高い表現力と引き換えに注意深い運用を求められる．なぜならメタプログラミングを用いたプログラムはバグを含みやすく，そしてエラーの発生場所とエラー原因が離れているためにバグ修正の難しいケースが多く存在するためである．例えば，C++ のテンプレートはテンプレートパラメータとして与えた型や値に応じたコードを生成する機能であるが，設計者の意図していないテンプレートパラメータが与えられた場合，コンパイラはテンプレートパラメータを与えた場所ではなくテンプレートの展開に失敗した位置に基づいたエラーメッセージを生成してしまう．また，C#の動的型生成 API を用いて実行時に不正な型を生成した場合，型構築時の不正な操作の場所ではなく型の読み込みやキャスト演算を行った場所でエラーが発生してしまう．これらを踏まえ，第 2.1 節ではメタプログラミングの例についていくつか紹介し，第 2.2 節ではメタプログラミングを利用しやすくするための研究やエラーメッセージの改善を目的とした研究を紹介する．第 2.3 節ではリフレクションおよび C#における動的型生成について解説し，続く第 2.4 節では既存の動的型生成 API が抱える課題に触れる．

2.1 メタプログラミング

メタプログラミングとはプログラム自体を操作対象とするプログラミング技法の総称である．メタプログラミングには実行時手続きの結果を処理系に反映させる実行時リフレクション，コンパイル時のテンプレート展開によって生成されたソースコードを他のソースコードに結合することでコードを最適化するテンプレートメタプログラミング，与えられた文字列をプログラムソースコード片として解釈し実行する eval などいくつかの種類が存在する．

2.1.1　3-LISP のリフレクション

3-LISP[1] はリフレクションを実装した最初期の言語として知られる．3-LISP における評価器は評価対象の式・評価環境・継続の組を用いて式の評価を行いその結果を継続に渡すように設計されているが，lambda-reflect を用いることで評価器の動作を変更し，標準の評価器の代わりに lambda を用いることが可能である．このようにして，3-LISP ではプログラマによるある種の文法拡張が可能となっている．

2.1.2　Smalltalk のリフレクション

Smalltalk は通常のプログラムとメタレベルのプログラムの境界が曖昧な言語であり [2]，リフレクションを始めとした様々なメタプログラミングが可能である．例えば，クラスの定義は実行環境へのクラスオブジェクトの登録として行われるほか，メソッドの登録もクラスオブジェクトへのメッセージ送信によって行われる．さらに，Smalltalk においてはその実行環境自体が大域変数 Smalltalk に格納されている．

2.1.3　C#のリフレクション

C#にもリフレクションの機能は備わっているが，クラスベースのオブジェクト指向静的型付き言語である点が 3-LISP や Smalltalk と大きく異なる．C#のリフレクション API は実行時のプログラムの情報を取得するものであり類似の API は Java にも存在するが，C#の API には実行時のコード生成や型生成を行う機能も備わっている．C#は型安全に設計されているが，この機能を介して動的に生成された型は型安全性が保証されない．

2.1.4　C++ のテンプレートメタプログラミング

C++ のテンプレートはテンプレートパラメータによって抽象化されたコードをコンパイル時に展開する機能であり，テンプレートメタプログラミングはこの機能を応用しコンパイル時に様々な計算を行う手法のことを指す．コンパイル時に計算を行うことで実行オーバーヘッドの最適化が可能になる一方，コンパイルに時間がかかる，プログラムのサイズが大きくなる，不正なテンプレートパラメータによるテンプレート展開失敗時のエラーメッセージが難解であるなど欠点もある．

2.1.5　MetaOCaml

MetaOCaml[3] は OCaml に多段階計算（multi-stage programming）の機構を導入したものである．多段階計算とは，静的な型システムの支援によって実行時のコード生成やコンパイル，評価などの操作を型安全に行うパラダイムである [4, 5]．MetaOCaml では，ブラケット.<および.>で囲まれた部分のコードは評価されずコードのままとなる．これをステージ化

（staging）と呼ぶ．ブラケットの内部のみに出現するエスケープ．~はコード生成時に既存のコードの一部を取り出しその場に展開するものである．これらの機能を組み合わせることで，MetaOCaml は静的型付けされたコード断片をコンパイル時に部分評価し，実行時にコードを組み合わせ処理を動的生成し実行する機構を実現している．

2.1.6 Template Haskell

Template Haskell は Haskell[6] の拡張機能であり，抽象構文木の操作によるコンパイル時メタプログラミングを実現する [7]．抽象構文木には Haskell の型が付いているため構文木の操作そのものは型安全に行われるが，構文木が表すコードの内容自体には具体的な型が付いていないため，型に問題のあるコードが生成された場合はエラーが生じる．なお，GHC 9.0 以降ではこの仕様に破壊的な変更が行われ，コード片にはコンパイル時に静的な型が付けられるようになった．また，Template Haskell をさらに拡張しステージ化機構の導入を試みた研究 [8] が存在する．

2.2 メタプログラミングを取り巻く環境の改善

第 2.1 節で取り上げたようにメタプログラミングには様々な種類があり，言語によってその実装は大きく異なっている．しかしこれらにはその高い表現能力と引き換えにコードの可読性が低い，開発時に生じるバグの修正が難しいといった共通の課題がある．この問題に対処するため，現在まで様々な研究や提案がなされてきた．ここではいくつかの例を紹介する．

concepts[9] は，C++ のテンプレートに対する改善案である．C++ のテンプレートは汎用的な処理の記述に適しているため標準ライブラリ内でも多用されているが，テンプレートパラメータとして不適切な値や型を指定した場合，そのミスの内容とは直接関係のない部分例えばテンプレートクラスライブラリ内などでテンプレート展開失敗による大量のコンパイルエラーが発生する可能性がある．この問題は Standard Template Library（STL）が C++ の標準ライブラリに組み込まれた 2003 年以前から知られており，エラーメッセージ改善を目的として concepts が提案された [10]．度重なる実装の延期を経て C++20（2020 年）より正式に C++ に取り込まれた concepts は，テンプレート型パラメータが満たすべき条件の集合によってテンプレートパラメータを制約する機能である．テンプレートに誤ったパラメータを与えてしまった場合，concepts を用いればエラーメッセージが concepts の制約条件を違反したことによるものとなるため，テンプレート展開に失敗した部分が報告される従来のエラーメッセージと比較して可読性が高くなり，バグの修正が容易になる．

プログラム 2.1 に concepts を用いない従来の方法でテンプレートを利用する場合の例を，プログラム 2.2 に同様の処理を concepts を用いて記述した場合の例を示す．

```
1 auto my_sort(auto c)
2 {
3     std::sort(c.begin(), c.end());
```

```
4 }
5
6 int main()
7 {
8     std::list<int> my_list{10, 20, 60, 40};
9     my_sort(my_list);
10 }
```

プログラム 2.1. 従来のテンプレートによる場合

```
1 int main()
2 {
3     std::list<int> my_list{10, 20, 60, 40};
4     std::ranges::sort(my_list);
5 }
```

プログラム 2.2. concepts を用いた場合

この 2 つの例はいずれも双方向連結リストクラス std::list のインスタンス lst を sort 関数によって並び替えることを意図したものだが，std::sort ならびに std::ranges::sort は RandomAccessIterator を要求する仕様であるため，std::list に対してこの関数を用いるとコンパイルエラーが生じる．g++ 11.1.0 において，プログラム 2.1 の例ではコンパイル時に 1265 行にわたるエラーメッセージが出力されたのに対し，プログラム 2.2 の例で出力されたエラーメッセージはわずか 36 行であり，さらにエラーの原因が std::ranges::sort 関数に渡されたテンプレートパラメータが concepts による制約条件を満たしていないためであることがエラーメッセージ内で明示されていた．ソースコードの見た目に大きな差がないのにもかかわらずこのような大きな差が生じたのは，std::sort 関数が従来のテンプレートのみで記述されたものであるのに対し std::ranges::sort 関数は std::random_access_iterator concept によってテンプレートパラメータが制約されているためである．

Hummingbird[11] は，動的型付き言語である Ruby に静的型付けの性質を与える手法を提案したものである．Ruby にはメタプログラミングによってクラスやメソッドの生成や書き換えなどを行う，Smalltalk に類似した機能が備わっているため，静的に動作する解析器によってコードの安全性を保障することが困難である．そこで Hammingbird は型情報を動的に追跡し，メソッド呼び出しのタイミングでその型安全性を実行前に検査する．文献内ではこれを just-in-time static type checking と呼んでいる．

衛生的マクロ（hygienic macros）は，マクロが含む識別子を適当にリネームし，マクロ展開に伴う識別子の衝突を回避することで識別子の誤った束縛を防ぐことが出来るようになっているマクロである [12]．衛生的マクロがサポートされている言語の例として Scheme が挙げられる [13] ほか，Rust のマクロも衛生的である．そもそもマクロとは，ある一定のルールに基づいてプログラムに対する文字列変換や構文変換を行う機能のことを指す．LISP のマクロは後者に相当するもので，LISP プログラムそのものが S 式であることを利用し，S 式を別の S 式へと変換することで構文変換を実現する．伝統的な LISP のマクロは衛生的でないため，マ

クロ展開によってマクロ外部の識別子が束縛され，予期せぬ動作をする場合がある．プログラム 2.3 は，伝統的な LISP マクロを用いた Scheme のプログラムである．この例で定義したマクロ swap-if!は 1 つめの引数が真（#t）である場合のみ 2 つめと 3 つめの引数を入れ換える．

```
(define-macro (swap-if! test left right)
    `(when ,test
           (let ((tmp ,left))
                (set! ,left ,right)
                (set! ,right tmp))))

(define (main args)
    (let ((x 100) (y 420) (when 10000))
         (begin
             (swap-if! #t x y)
             (display x)
             (newline)
             (display y))))
```

プログラム 2.3. 伝統的な LISP マクロによる外部識別子の予期せぬ束縛

一見問題なく動作するように見える swap-if!だが，伝統的なマクロはパターンによる識別子の置き換えに過ぎないため，マクロ内で用いた識別子がマクロ作成者の意図するものと異なるものに置き換わっていた場合，正常に動作しない．この例では，swap-if!の呼び出し前に識別子 when が定義されているため，展開された swap-if!内の when は when 構文ではなくローカル変数を束縛してしまう．

プログラム 2.4 は，衛生的マクロを用いた Scheme のプログラムである．この例で定義したマクロ hygienic-swap-if!はプログラム 2.3 の例と同様に動作する．

```
(define-syntax hyginic-swap-if!
    (syntax-rules ()
        ((_ test left right)
            (when test
                  (let ((tmp left))
                       (set! left right)
                       (set! right tmp))))))

(define (main args)
    (let ((x 100) (y 420) (when 10000))
         (begin
             (hygienic-swap-if! #t x y)
             (display x)
             (newline)
             (display y))))
```

プログラム 2.4. 伝統的な LISP マクロによる外部識別子の予期せぬ束縛

衛生的マクロ内では，登場した識別子が外側のスコープの影響を受けることなく常にマクロのスコープから見えるグローバルな識別子を参照出来るように扱われる．そのため hygienic-swap-if!はこの例のように when が外側のスコープで束縛されている場合でもマクロ設計者の意図通り正常に動作する．

2.3　実行時リフレクションと動的型生成

実行時リフレクションとは，実行時の処理を言語処理系の動作に反映させる技術の総称である．実行時リフレクションの実現方法は言語によって多岐に渡るが，ここでは特に C#におけるリフレクションについて説明する．C#におけるリフレクションは，プログラムのメタデータを実行時に取り出して利用する機能として実現されている．C#のリフレクション API でサポートされている操作の例を以下に挙げる．

1. 実行環境に存在する型の一覧の取得
2. 型が持つメソッド，フィールド，プロパティなどの情報の取得
3. 上記の操作によって得たメタ情報を用いたメソッド呼び出しなどの操作
4. 型やメソッドが持つ属性（Attributes）の取り出し
5. Expression Tree による実行時の式木操作およびメソッドとしての実行
6. System.Reflection および System.Reflection.Emit 名前空間が提供する API による動的な型生成および生成した型の.dll ファイルへの出力

これらの処理のうち，コンパイル時の型環境に存在しない型を実行時に扱う点において動的型生成は特異である．C#のリフレクション API は特殊な構文ではなく通常のクラスやメソッドとして実装されており，動的型生成は AssemblyBuilder，ModuleBuilder，TypeBuilder，MethodBuilder などのクラスを用いて以下の手順で行う．

1. TypeBuilder のインスタンスを生成する．このインスタンスは生成しようとしている型の「原型」を表す
2. TypeBuilder の持つ DefineMethod や DefineMethod などのメソッドを呼び出すことで TypeBuilder が表す型の「原型」を修飾し，目的の型の「原型」を作る
3. CreateType メソッドを用い，TypeBuilder のインスタンスから Type 型のインスタンスを得る
4. 得られた Type 型のインスタンスを利用して処理を行う

Type 型は型情報を表すクラスであり，通常の型についても typeof 演算子を利用することでその型情報が取得出来るようになっている．C#の標準ライブラリには Type 型のインスタンスを引数にとりその型のインスタンスを生成する Activator.CreateInstance メソッドがあるため，動的に生成された型のインスタンスは基本的にこのメソッドを用いて利用される．

プログラム 2.5 は，C#の標準ライブラリを用いて実行時に動的にクラスを生成するプログラムの例である．

```
1  // PlayerGenerator.cs
2  var typeBuilder = modBuilder.DefineType(charData.Name);
3  var walkMethod = typeBuilder.DefineMethod("Walk",
4          Virtual | Public, typeof(void),
5          new[] { typeof(InputStates) });
6  /* definition of Walk method (omitted) */
7  var jumpMethod = typeBuilder.DefineMethod("Jump",
8          Virtual | Public, typeof(void),
9          new[] { typeof(InputStates) });
10 /* definition of Jump method (omitted) */
11 typeBuilder.SetParent(typeof(Player));
12 var player_t = typeBuilder.CreateType()!;
13
14 // PlayerLoader.cs
15 var player = (Player)Activator.CreateInstance(player_t)!;
```

プログラム 2.5. 実行時型生成のサンプルコード

このコードは，ゲームのユーザーが記述した設定ファイルをもとにキャラクターの動作の内容を動的に生成するシーンを想定した，ゲームプログラムの疑似コードである．このプログラムでは，クラスの定義 (typebuilder) を生成し，そこに Walk メソッドおよび Jump メソッドの定義を追加している．そして，Player クラスを基底クラスに指定し，CreateType メソッドの呼び出しによって型を生成している．このようにして生成された型のインスタンスは，型の生成とは別のタイミングで CreateInstance メソッドによって利用されている．

2.4 動的型生成の課題

前述の通り，動的に作られた型はコンパイル時の型環境に存在しないため，静的な型検査を行われないまま実行環境に存在出来てしまう．これらの型は CreateType メソッドによる生成時に最低限の整合性チェックを受けるが，このチェックを通過したからといって型が C#において安全なものであるとは限らない．また，実行時エラーは生じないが開発者の意図と異なる動作をする型を生成することも出来てしまい，そのような場合はバグの原因を特定するための情報が乏しいことからバグの修正が困難となる．例えば，動的型生成 API を用いることでプログラム 2.6 のようにシグネチャと名前が完全に一致するメソッドを同じ型に複数宣言することが出来てしまうが（この例ではループによりメソッド Function を 100 つ定義している），このような型は CreateType メソッドによるチェックではエラーにならない．これは，C#の動作環境である.NET が同名同シグネチャのメソッドの多重定義を許容しているためである．このような多重定義されたメソッドを呼び出した場合同名メソッドのうち最初に定義されたものが呼び出されるが，開発者のミスによって意図せぬ部分でメソッドの定義が重複していることに気づくのは難しく，発見されないバグを混入させる原因ともなる．

```
1     public interface IActor
2     {
3       void Function();
4     }
5     ...
6     var typeBuilder = modBuilder.DefineType("DynamicActor",
          TypeAttributes.Class,typeof(object));
7     for(var i=0;i<100;i++)
8     {
9         var methodBuilder = typeBuilder.DefineMethod("Function",
              MethodAttributes.Public | MethodAttributes.Virtual,
              typeof(void),Type.EmptyTypes);
10        var exp = Expression.Block(
11          Expression.Call(typeof(Console).GetMethod(name: "WriteLine
                ",types: new Type[]{typeof(string)})!),Expression.
                Constant("Hello!"));
12        Expression.Lambda(exp).CompileToMethod(methodBuilder);
13    }
14    typeBuilder.AddInterfaceImplementation(typeof(IActor));
15    var t = typeBuilder.CreateType()!;
16    var dy = (IActor)Activator.CreateInstance(t)!;
17    dy.Function();
```

プログラム 2.6. メソッド定義の重複によるバグ

また，プログラム 2.7 の例ではメソッドの抽象性と基底クラスの指定に誤りが存在するが，この誤りも実行時に初めて検出される．

```
1  // correct
2  var jumpMethod = typeBuilder.DefineMethod("Jump",
3          Virtual | Public, typeof(void),
4          new[] { typeof(InputStates) });
5  typeBuilder.SetParent(typeof(Player));
6
7  // error
8  var jumpMethod = typeBuilder.DefineMethod("Jump",
9          Public, typeof(void),
10         new[] { typeof(InputStates) });
11 typeBuilder.SetParent(typeof(Enemy));
```

プログラム 2.7. メソッド定義の重複によるバグ

この誤りを含むプログラムを実行した場合，以下のような実行時エラーメッセージが表示される．

```
1  Unhandled exception.
```

```
2 System.InvalidCastException: Unable to cast object of type '
      DynamicPlayer' to type 'Player'.
3   at PlayerLoader.Load(String name)
4   in PlayerLoader.cs:line 8
```

動的型生成に起因する型エラーは必然的に実行時エラーとして検出されるが，この種のエラーメッセージはあくまで実行時に発生した問題を指摘するものであり型生成時の問題を指摘するものではないため，エラーメッセージで指摘された部分を確認するだけではエラー原因の特定が難しい．この問題は C++ におけるテンプレート展開失敗によるコンパイルエラーに類似しているが，concepts の導入によって問題が軽減した C++ とは異なり C#の動的型生成 API にはこの問題を解決する手段が提供されていない．

2.4.1 実行時エラーの例

動的型生成に起因する実行時エラーにはいくつかの種類がある．この節ではこれらのエラーのうち典型的なものの例をサンプルコードを用いて紹介する．

プログラム 2.8 は，インタフェース型 IActor を実装したクラスを動的に生成することを目的とした，誤りを含む C#のプログラムを簡略化したものである．動的に生成する型にインターフェースの実装を与えるためには AddInterfaceImplementation メソッドを利用する必要があるが，このコードはメソッド呼び出しを行っていないため生成される型が IActor との互換性を持たない．このコードを実行すると，生成したクラスを IActor 型へキャストする部分で実行時例外 InvalidCastException が発生する．

```
1     public interface IActor
2     {
3       void Function();
4     }
5     ...
6     var typeBuilder = modBuilder.DefineType("DynamicActor",
          TypeAttributes.Class,typeof(object));
7     var methodBuilder = typeBuilder.DefineMethod("Function",
          MethodAttributes.Public | MethodAttributes.Virtual,typeof(
          void),Type.EmptyTypes);
8     var exp = Expression.Block(
9       Expression.Call(typeof(Console).GetMethod(name: "WriteLine",
            types: new Type[]{typeof(string)})!),Expression.Constant("
            Hello!"));
10    Expression.Lambda(exp).CompileToMethod(methodBuilder);
11    var t = typeBuilder.CreateType()!;
12    var dy = (IActor)Activator.CreateInstance(t)!;
13    dy.Function();
```

プログラム 2.8. interface 実装忘れの例

プログラム 2.9 はフィールド x および y を持つクラス Vector2 を生成する目的で誤ってフィールド x を 2 つ持つクラスを生成した C#プログラムを簡略化したものである．

```
1    var typeBuilder = modBuilder.DefineType("Vector2",TypeAttributes
         .Class,typeof(object));
2    var xBuilder = typeBuilder.DefineField("x", typeof(int),
         FieldAttributes.Public);
3    var yBuilder = typeBuilder.DefineField("x", typeof(int),
         FieldAttributes.Public);
4    var vec2 = typeBuilder.CreateType()!;
5    var v = Activator.CreateInstance(t)!;
6    // 以下の行で例外発生
7    v.x = 100;
```

プログラム 2.9. フィールド x を 2 つ持つクラスを生成する例

このプログラム内で生成されるクラス Vector2 は，C#では表現出来ない型である．通常の C#プログラムでは，同名フィールドを複数持つクラスの定義はコンパイルエラーになるためである．7 行目において発生する実行時例外 AmbiguousMatchException は複数存在するフィールド x の曖昧性がエラーの直接的な原因であることを示しているが，本来この種のエラーはフィールドアクセス時ではなく型生成時に検出されるべきである．また，この例外によるエラーメッセージだけでは実行時にフィールド x を型に追加した部分を特定することが出来ないため，バグ修正のためソースコードを網羅的に確認する必要が生じる．

2.4.2　実行時エラーの分類

型生成を行うリフレクション計算に起因する実行時エラーは，その発生タイミングにより以下のように分類出来る．

1. **CreateType 呼び出し時，生成しようとした型に明らかな不備がある場合**
   CreateType メソッドは，型を表す TypeBuilder のインスタンスから Type 型のインスタンスを生成するメソッドである．このメソッド内部では TypeBuilder の持つ情報が参照され，.NET の型としての整合性が確認された後に Type 型のインスタンスが生成される．この種のエラーの例として，実装したインタフェースが要求するメソッドを生成した型が持っていない場合やメソッドの本体が欠落している場合などがある．
2. **生成した型のインスタンスを既存の型にキャストしたとき，キャスト先の型との間に互換性がない場合**
   先述の例のようにキャスト先のインタフェース型を AddInterfaceImplementation メソッドによって指定していない場合や継承すべき基底クラスが指定されていない場合が該当する．発生する例外は InvalidCastException である．
3. **C#の型として不適格な型のインスタンスのメンバーにアクセスした場合**
   メソッドやフィールドの多重定義など，C#から利用した際にエラーとなる型を生成し

た場合が該当する．

これらのエラーのうち 3 つめのエラーは API 設計の見直しによって CreateType 時に検出することが可能になるため，動的に生成された型を検査する必要があるのは事実上 CreateType メソッドの呼び出し時と既存の型へのキャスト時の 2 箇所に限られる．

以上で繰り返し述べたように，C#におけるリフレクション API を用いた動的型生成はバグを含みやすいものであり，また発生したエラーの原因を特定することが困難である．それにもかかわらず現在においてもこの問題が解決されていない理由はおそらく以下の 2 点であると推測される．

- 動的な型生成を行う機能は C#の言語機能全体から見れば大きなものではなく，言語機能を拡張することにより得られる恩恵が機能拡張のコストに見合わないため問題解決の優先度が低く見積もられている
- 動的型生成処理に限らず，一般にリフレクション計算はバグを含みやすいものであると認知されているが，それ故にリフレクションを用いずに実現可能なことは性能を犠牲にしてでもリフレクションを避けて実現することが推奨されてきたため，問題解決のモチベーションが低い

エラーメッセージのわかりにくさから同様に扱いが難しいとされる C++ のテンプレートは，言語機能の大きな部分を占めている点，標準ライブラリ内で多用されていることからテンプレートの利用を避けたプログラミングが事実上不可能な点などから改善の需要が極めて高く，C#とは状況が大きく異なった．一方 C#の動的型生成は問題そのものの回避により問題を「解決」してきた．しかし，これらの問題は本来何らかの方法で解決すべきである．

2.4.3 難しさの要因

実行時例外の発生場所がその根本原因である操作の場所と離れている場合，エラーの修正は容易でない．なぜなら，例外の発生原因を特定するためには例外が発生する以前に行われた処理を確認し原因となる処理を探し出す必要があるためである．この作業にはソースコードを目視で確認する，プログラム実行中の変数や式の値をログファイルや標準出力に出力し確認する，デバッグツールを利用し実行中のプログラムの状態や実行経路を確認するなどの方法が用いられるが，特に大規模なプログラムにおいてこれらの作業はコストが大きい．

そもそも実行時例外は，何らかの原因で処理が失敗した場合や処理に必要な前提条件が満たされていない場合に処理系や各言語の例外発生機構によって生成されるものである．例えば C#では，ゼロ除算を実行した場合処理系によって `DivideByZeroException` 例外が投げられ，コレクションクラス（配列のように同じ型の要素を複数格納出来るもの）のインスタンスを先頭から探索し条件を満たす最初の要素を返すメソッド `Enumerable.First` は条件を満たす要素が存在しない場合 `InvalidOperationException` 例外を発生させる．

C#における動的型生成に起因する実行時例外は型生成およびその直後に行われる型の読み

込み時に発生するが，その根本原因は型構築時の不正な操作にあり，エラーの発生場所とは離れている．このことが，動的型生成を含むプログラムのデバッグをより難しくしている．

第 3 章

エラーメッセージの改善

エラーの原因となりうる操作の履歴をその実行時に記録し，エラーの発生時その情報から推測した根本原因の位置を実行時エラーメッセージ内に含めることによって，エラーの根本原因と実行時エラーの発生箇所が離れている場合のバグ修正を容易にする手法を提案する．さらに動的型生成を行うリフレクション計算を原因とする実行時エラーの発生時，あらかじめ記録された操作の履歴と型のもつ不正な性質を元にエラー原因の分析を行い，元のプログラム内におけるエラーの原因となる誤りを含む可能性のある場所をエラーメッセージ内で指摘することによりバグの原因の早期発見が可能な動的型生成ライブラリを開発した．

提案手法の実現方法には，大きく分けて処理系への機能追加を行うアプローチと既存の処理系上で動作させるアプローチの 2 種類が存在する．処理系への機能追加を行うアプローチの場合，型構築操作の履歴を処理系が直接扱うことで，標準ライブラリの動的型生成 API を用いた既存のプログラムでも詳細なエラーメッセージを表示させることが可能になる．しかしこの方法は，既に広く使われている.NET の処理系に新たな機能を追加する必要があることから現実的ではない．一方，既存の処理系の上で動作させるアプローチの場合，標準ライブラリの代わりに新たに開発されたライブラリを用いてプログラムを記述する必要があるため，既存のコード資源をそのまま流用することは出来ない．しかし，処理系を改造することなく利用できることから，実現可能性は高い．これらを踏まえ，本研究では既存の処理系の上で動作することを前提としてエラーメッセージの改善を実践した．

動的型生成に起因するエラーの多くは生成された不正な型の読み込み時に発生する実行時例外であり，型生成時の検査によって誤りが発見されているわけではないため型が含む誤りによる分類は行われない．一方，同様の型エラーを含む型を通常の C#プログラムで記述した場合，コンパイラはそのエラーの原因に基づいて分類された詳細なコンパイルエラーメッセージを出す．このエラーメッセージはエラー修正に有用であるため「良い」エラーメッセージであると言える．そこで我々は，動的型生成により生じた実行時エラーメッセージがコンパイル時の型エラーメッセージと同等の情報を含むようにすることでエラーメッセージを改善した．エラー原因となりうる操作の位置の取得には C#5.0 で追加された Caller Info 機能を用いた．エラーメッセージの改善にあたり，動的型生成により生じる型エラーをその原因により分類し，コンパイル時の型エラーと対応付けた．また，対応するコンパイルエラーの存在しない実行時エ

ラーについても独自に分類を行った．3.1 節では提案手法について，3.2 節では提案手法に基づいて開発したライブラリの仕様について，3.3 節ではライブラリの内部実装について詳しく説明する．

3.1　操作履歴の記録

機械学習ライブラリにおけるモデルの構築をはじめ，API 呼び出しの繰り返しによってデータを構築するよう設計されているライブラリは数多く存在する．このようなライブラリの利用時には，API 呼び出しの時点では誤りであると断定出来ない操作が原因となり後の処理が実行時エラーを発生させるケースが存在する．このようなケースはエラーの発生場所とその原因が離れているためデバッグが難しい．しかし，もし実行時エラーメッセージがその原因の位置を指摘するようになっていれば，デバッグはより容易になる．API 呼び出しが行われるたびにその内容とソースコード内における位置を記録し，エラー発生時に過去の操作の中からエラー原因と考えられるものを特定する仕組みを実装することで，これは実現できる．ただし，過去の操作履歴と実行時エラーの内容から原因を特定する方法そのものはライブラリごとに個別に考える必要がある．

ここで，簡単な例を紹介する．プログラム 3.1 は，入力値とそれに対応する処理のペアを登録し，後に入力データを与えて処理を呼び出すプログラムの疑似コードである．

```
1 var mapper = new Mapper();
2 mapper.Register(1, () => Console.WriteLine("Hello"));
3 mapper.Register(4, () => Console.WriteLine("input is 4."));
4 mapper.Register(7, () => Console.WriteLine("Good morning."));
5 mapper.Register(2, () => Console.WriteLine("Now Loading..."));
6 mapper.Register(3, () => Console.WriteLine("Complete!"));
7 mapper.Register(4, () => Console.WriteLine("Good bye."));
8 mapper.Call(4);
```

プログラム 3.1. エラーと原因が離れている例

この例では，3 行目と 7 行目で同じ入力値に対する処理を登録している．このプログラムを実行した場合，ライブラリの仕様によって以下の 3 種類の結果が考えられる．

1. 7 行目でエラー発生
2. 8 行目でエラー発生
3. エラーが発生しない

1. の場合，7 行目でエラーが発生するのは登録しようとしたキーが 3 行目のものと同一であるためである．Mapper クラスが標準ライブラリの Dictionary クラスのラッパーとして設計されていた場合はこの結果となる．このとき，実行時エラーメッセージは以下のようになる．

```
1 System.ArgumentException: An item with the same key has already been
      added. Key: 4
```

```
2     at System.Collections.Generic.Dictionary`2.TryInsert(TKey key,
          TValue value, InsertionBehavior behavior)
3     at System.Collections.Generic.Dictionary`2.Add(TKey key, TValue
          value)
4     at Mapper.Register(Int32 key, Action action) in Mapper.cs:line 4
          at Program.<Main>$(String[] args) in Program.cs:line 7
```

例外が 7 行目で発生したこと，キーの競合が原因であることがエラーメッセージから確認出来るが，以前同じキーを登録したのがどこであるかはこのメッセージからはわからない．

2. の場合，キー 4 に対応する操作が複数登録されていたことによる曖昧性が原因でエラーが生じる．Mapper クラス内のデータ構造がキーの重複を許容する設計であり，かつ処理の呼び出し時にキーの重複を確認する仕様である場合はこの結果となる．このとき，実行時エラーメッセージは以下のようになる．

```
1 System.InvalidOperationException: Sequence contains more than one
      element
2     at System.Linq.ThrowHelper.ThrowMoreThanOneElementException()
3     at System.Linq.Enumerable.TryGetSingle[TSource](IEnumerable`1
          source, Boolean& found)
4     at System.Linq.Enumerable.Single[TSource](IEnumerable`1 source)
5     at Mapper.Call(Int32 key) in Mapper.cs:line 12
6     at Program.<Main>$(String[] args) in Program.cs:line 8
```

登録された操作が複数あるためにエラーが生じたこと，エラーの直接の原因となったメソッド呼び出しはエラーメッセージから確認出来るが，要素の重複が生じた場所はこのメッセージからはわからない．

3. の場合，ライブラリが重複するキーの多重登録を認める設計になっていると考えられる．登録された処理の呼び出し時に重複する要素全てが呼び出される仕様であるならばこの設計は妥当であるが，そうでない場合はおそらく最初に登録された処理か最後に登録された処理のどちらか一方のみが呼び出される設計であるため，1. や 2. の場合以上に開発者の意図せぬ挙動を取る可能性が高い．

いずれの場合においても，通常の設計ではエラーの発生位置とエラーメッセージから原因となった操作を特定することは容易でない．しかし，Mapper クラスが Register メソッドの呼び出し履歴を保持する設計になっていれば，キーの重複を検出した際に操作履歴から同じキーを登録した操作の位置を特定することが可能になる．エラーメッセージの生成にこの情報を利用することで，開発者は実行時エラーの根本原因を容易に知ることが出来る．参考として，この機能を実装した Mapper クラスで同様のプログラムを実行した場合のエラーメッセージを以下に示す．

```
1 System.ArgumentException: An item with the same key has already been
      added. key: 4
2         at <Main>$ in Program.cs:line 3 as Register(4,() => Console.
```

```
            WriteLine("input is 4."))
3    at Mapper.Register(Int32 key, Action action, Int32 line, String
         memberName, String path, String keyExpression, String
         actionExpression) in Mapper.cs:line 25
4    at Program.<Main>$(String[] args) in Program.cs:line 7
```

エラーの発生位置が 7 行目であること，キーの重複がエラー原因であること，同じキーを登録した位置が 3 行目であることがエラーメッセージ内で全て示されており，エラーメッセージの質が向上していると言える．

3.2　開発したライブラリの仕様

C#標準ライブラリが提供する動的型生成 API の仕様を見直し，新たな API 仕様を提案する．さらにクラスのインスタンスメソッドに対応した動的コード生成 API を提案し，この 2 つを組み合わせて動的型生成をより扱いやすいものにする．API 仕様の説明にあたり，先に 3.2.1 節で生成対象である.NET の型システムについて説明する．その後 3.2.2 節で標準ライブラリの API の仕様に触れ，その問題を解決した新たな API を 3.2.3 節で提案する．

3.2.1　共通言語基盤と共通型システム

C#の動作環境である共通言語基盤（Common Language Infrastructure/CLI）では，コンパイルされたプロセスやライブラリのことをアセンブリと呼ぶ．アセンブリには実行コードの他エントリポイントの情報や型情報などのメタデータ，リソース等が含まれている．アセンブリは複数のファイルで構成することが可能であり，その構成要素はモジュールと呼ばれ，ひとつひとつのモジュールはクラス，構造体，インタフェース等の型およびグローバルメソッドとグローバルフィールドからなる．

.NET の型システムは共通型システム（Common Type System/CTS）と呼ばれる．.NET 言語である C#の型システムはこの共通型システムに等しい．共通型システムにおける型は，オブジェクトが実際の値によって表される値型とオブジェクトがその参照によって表される参照型に分類される．共通型システムの型にはクラス，構造体，列挙型，インタフェース，デリゲートの 5 つのカテゴリが存在し，このうち構造体と列挙型が値型，クラスとインタフェースとデリゲートが参照型である．クラスおよび構造体は複合型であり，その構成要素はフィールド，プロパティ，メソッド，コンストラクタおよびイベントである．クラスでは他のクラスを継承して派生クラスを定義することが可能だが，その他のカテゴリの型には暗黙的な派生元のクラスが存在し，ユーザ定義の派生型を作ることは出来ない．ただし，インタフェースの実装はクラスの継承とは異なるため，構造体でも可能である．これらの言語要素には，属性（attribute）を用いることによってユーザ定義のメタデータを追加できる．属性はそれ自体が System.Attribute クラスを継承したクラスであり，通常のクラスと同様にユーザが定義出来る．

その他に，型やその構成要素にはアクセシビリティと呼ばれる特性がある．アクセシビリティは他の型からのアクセス可能性を制御するための修飾子であり，型やメンバはアクセシビリティで認められた範囲からしかアクセスすることが出来ない．例えば，private なメソッドはその型以外のメソッドからはアクセス出来ない．

3.2.2 System.Reflection.Emit の仕様

C#のリフレクション API は標準ライブラリの System.Reflection 名前空間で提供されており，そのうち動的型生成に関係するものは System.Reflection.Emit 名前空間にある．リフレクションによる動的型生成では，型の定義の前にその型の所属先となるアセンブリとモジュールを定義する必要がある．アセンブリの定義には AssemblyBuilder 型のインスタンスを返す AssemblyBuilder.DefineDynamicAssembly が，モジュールの定義には AssemblyBuilder 型のメソッドで ModuleBuilder 型のインスタンスを返す DefineDynamicModule メソッドが用いられる．多くの場合，1 つのライブラリないしアプリケーション内で複数の型を動的に生成し利用する場合であっても，動的アセンブリや動的モジュールは 1 つしか用いない．なぜなら，アセンブリやモジュールの分割は分割コンパイルのために行われるものであり，その場で型を利用する際には不必要なためである．

動的アセンブリと動的モジュールを定義することで，動的な型が定義出来るようになる．動的な型の構築に用いる TypeBuilder は ModuleBuilder の DefineType メソッドによって得られる．DefineType にはいくつかのオーバーロードが存在するが，その全てにおいて第 1 引数として型名の文字列を取る．その他，インタフェース型であるかどうかやアクセシビリティの情報が格納された TypeAttributes（これは前述の属性とは無関係である），基底クラスを表す Type などを引数に取るオーバーロードが存在する．なお，構造体型を構築する場合は構造体の共通基底クラスである ValueType を基底クラスとして指定する必要がある．

TypeBuilder にはその構成要素を定義するためのメソッドが備わっている．これらはそれぞれ Define***という共通の規則で命名されており，それぞれが対応する***Builder のインスタンスを返す．以下にその一覧を示す．なお，DefineGenericParameters および DefineMethodOverride を除く全ての Define***メソッドは記載した引数以外に対応する ***Attributes も引数に取る．

- **DefineConstructor**
  引数型の配列を引数に取り，コンストラクタ定義を表す ConstructorBuilder を返す
- **DefineDefaultConstructor**
  MethodAttributes のみを引数に取り，デフォルトコンストラクタ定義を表す ConstructorBuilder を返す
- **DefineEvent**
  名前とイベントの型を引数に取り，イベント定義を表す EventBuilder を返す
- **DefineField**

名前と型を引数に取り，フィールド定義を表す FieldBuiler を返す

- **DefineMethod**
  名前，戻り値の型，引数型の配列を引数に取り，メソッド定義を表す MethodBuilder を返す．メソッド本体の定義は MethodBuilder のメソッドを用いて行う
- **DefineProperty**
  名前, 戻り値の型, 引数型の配列を引数に取り, プロパティ定義を表す PropertyBuilder を返す．.NET のプロパティの実体はプロパティへの代入時に呼び出される set メソッドと値の取り出し時に呼び出される get メソッドの組である．プロパティの構築する場合はプロパティ本体だけではなく set メソッドおよび get メソッドに相当するメソッドを型に定義し，PropertyBuilder の SetSetMethod と SetGetMethod を用いてこれらとプロパティを対応付ける必要がある
- **DefineGenericParameters**
  ジェネリック型パラメータの名前の配列を引数に取り，ジェネリック型パラメータを表す GenericTypeParameterBuilder の配列を返す．ここで得られた型パラメータはメソッドやフィールドの定義において型として利用出来る．
- **DefineMethodOverride**
  自身のメソッドを表す MethodInfo とそのメソッドがオーバーライドする基底クラスのメソッドの MethodInfo を引数に取り，メソッドのオーバーライドを定義する．MethodBuilder は MethodInfo の派生クラスであるため，第 1 引数にはメソッド定義に用いた MethodInfo のインスタンスを用いる場合が多い

これらのメソッドを用いて構成要素を定義した TypeBuilder は，CreateType メソッドによって.NET の型にすることが出来る．このようにして動的に生成された型へのアクセスは CreateType メソッドが返す Type 型のインスタンスを介して行われる．

CreateType メソッドは，呼び出し時の TypeBuilder が保持する情報を元に型を生成する．生成対象の型に問題がある場合そのほとんどは CreateType の呼び出し時に検出されるが，エラーメッセージはエラーの直接的な内容とその発生箇所の情報しか提供せず，型に何らかの不備があったことしかわからないケースも多い．以下に CreateType メソッドが投げる可能性のある実行時例外とその原因を示す．

- **InvalidOperationException**
  入れ子になっている型の定義であって外側の型が生成されていない場合，非抽象型であるのにもかかわらず抽象メソッドを含む場合，および抽象クラスやインタフェース型でないのに本体のないメソッドを含む場合に発生する
- **ArgumentException**
  型が含むいずれかのメソッド本体の IL が不正なラベルを含んでいる場合に発生する
- **NotSupportedException**
  型が含むいずれかのメソッド本体に無効な IL が含まれている場合に発生する
- **TypeLoadException**

生成した型のロードに失敗した場合に発生する

プログラム 3.2 は，メソッド Method のみを含むインタフェース IDummy を実装したクラス DynamicType を動的に生成する目的のコードである．

```
1     public interface IDummy
2     {
3         void Method();
4     }
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             var asmBuilder = AssemblyBuilder.DefineDynamicAssembly(
                  new("DynamicAssembly"), AssemblyBuilderAccess.Run);
10            var modBuilder = asmBuilder.DefineDynamicModule("
                  DynamicModule");
11            var typeBuilder = modBuilder.DefineType("DynamicType",
                  TypeAttributes.Class, typeof(object));
12            var methodBuilder = typeBuilder.DefineMethod("Method",
                  MethodAttributes.Public, typeof(void), Type.
                  EmptyTypes);
13            var ilGen = methodBuilder.GetILGenerator();
14            ilGen.Emit(OpCodes.Ret);
15            typeBuilder.AddInterfaceImplementation(typeof(IDummy));
16            var type = typeBuilder.CreateType();
17        }
18    }
```

プログラム 3.2. 情報に乏しいエラーメッセージが表示される例

このプログラムを実行すると，16 行目の CreateType メソッド呼び出し時に実行時例外が発生し，以下のエラーメッセージが表示される．

```
1 Unhandled exception. System.TypeLoadException: Method 'Method' in
      type 'DynamicType' from assembly 'DynamicAssembly, Version
      =0.0.0.0, Culture=neutral, PublicKeyToken=null' does not have an
       implementation.
2    at System.Reflection.Emit.TypeBuilder.CreateTypeNoLock()
3    at System.Reflection.Emit.TypeBuilder.CreateType()
4    at Program.Main(String[] args) in Program.cs:line 16
```

このエラーメッセージは 16 行目で呼び出された CreateType メソッドの内部で例外が発生したこと，そして生成した型 DynamicType がメソッド Method を実装していないため型のロードに失敗したことを示している．ところがソースコードを確認すると，12 行目におい

て Method は定義されており，13 行目および 14 行目でその本体は記述されているように見える．実は，このバグの原因は DefineMethod において第 2 引数の MethodAttribute に MethodAttributes.Virtual が含まれていないことである．.NET において型のメンバーがインタフェースのメンバーの実装となるためには public かつ virtual でなければならないが，C#ではインタフェースのメンバーと同名かつ同シグネチャのメンバーはアクセシビリティが public であれば暗黙的に virtual 指定した扱いとなるためこの仕様はあまり知られておらず，その上このエラーメッセージは 12 行目の DefineMethod の存在に一切触れないため，問題の特定が難しくなってしまっている．

比較対象としてプログラム 3.3 とそのコンパイルエラーメッセージを示す．

```
1 public interface IDummy
2 {
3     void Method();
4 }
5 class Dummy : IDummy
6 {
7     void Method() { }
8 }
```

プログラム 3.3. アクセシビリティ指定を忘れた例

このプログラム内のクラス Dummy は IDummy インターフェースを実装しているが，メソッド Method のアクセシビリティが private であるためにコンパイルエラーが発生し，以下のメッセージが表示される．

```
1 Program.cs(36,15): error CS0737: 'Dummy' does not implement
      interface member 'IDummy.Method()'. 'Dummy.Method()' cannot
      implement an interface member because it is not public.
```

このエラーメッセージはアクセシビリティが public でないために Dummy.Method が IDummy.Method の実装になれないことを指摘しており，開発者は容易にミスを修正することが可能である．

3.2.3　拡張 API

3.2.2 節で述べたように，標準ライブラリの API を用いた動的型生成処理がバグを含んでいた場合，エラーメッセージの情報は原因の特定にはあまり役立たない．我々は，バグの修正に役立つ実行時エラーメッセージには以下の要素が必要であると考えている．

1. 発生した問題の内容が含まれている
2. 発生した問題の直接的な原因が示されている
3. 発生した問題の原因となった誤りが指摘されている
4. エラーメッセージの量が適切である

標準ライブラリの API はこのうち 3. を欠いているため，バグの原因を特定するためにソースコードの広範囲を確認する必要がある．C++ のテンプレートは 4. を欠いており，大量のエラーメッセージからバグの修正に必要な情報を探す必要があったが，concepts の導入によってこの問題は軽減され，さらに 3. が充実した．

我々は，実行時エラーメッセージがその原因となる誤りの情報を含むように改良した，新たな動的型生成 API を提案する．この節では，提案する API のうちエラーメッセージの改善には直結しない部分について，既存 API からの変更点およびその理由を説明する．

提案する API では，型への構成要素の追加に Add***メソッドを用いる．これらのメソッドの一覧を以下に示す．

- **AddMethod**
- **AddField**
- **AddProperty**
- **AddIndexer**
- **AddInterfaceImplementation**
- **AddConstructor**

これらのメソッドは既存 API の Define***メソッドに類似しているが，型情報を通常の引数ではなくジェネリック型パラメータとして受け取るように設計されている．これにより，不完全な型情報を用いることによる実行時エラーを回避することが出来るようになった．プログラム 3.4 は，既存 API において動的メソッドの戻り値の型に不完全な型を割り当てた例である．

```
1    var methodBuilder = typeBuilder.DefineMethod("Method",
         MethodAttributes.Public | MethodAttributes.Virtual, typeof(
         List<>), Type.EmptyTypes);
2    var ilGen = methodBuilder.GetILGenerator();
```

プログラム 3.4. 引数が不完全型のメソッド定義

この MethodBuilder によって生成されるメソッドは List<> Method() というシグネチャを持つ．List<>はジェネリック型パラメータが指定されていないジェネリック型である．このような型は C#だけでなく.NET においてもインスタンス化出来ないが，このメソッドの生成は成功してしまう．なお，C#ではこのメソッドのシグネチャを表現することが出来ないため通常の方法による呼び出しは出来ない．リフレクションによってこのメソッドの MethodInfo を取得し呼び出した場合，ランタイムが System.BadImageFormatException 例外を出力して停止する．

以下のプログラム 3.5 はプログラム 3.4 と同様の処理を提案する API を用いて記述したものである．

```
1    var methodBuilder = typeBuilder.AddMethod<List<>>("Method",
         MethodAttributes.Public | MethodAttributes.Virtual);
2    var ilGen = methodBuilder.GetILGenerator();
```

プログラム 3.5. 新しい API における引数が不完全型のメソッド定義

このプログラムは C#の文法に違反するため，実行時エラー以前にコンパイルエラーとなる．

これらの Add***メソッドには，メソッド本体を追加の引数に取るオーバーロードが用意されている．これを用いることで***Builder クラスを介さずに型の構成要素が定義出来るようになる．既存 API において，メソッドの本体を記述する際にはまず DefineMethod メソッドによって MethodBuilder を取得し，その後 MethodBuilder から取り出した ILGenerator の Emit メソッドを繰り返し呼び出すことで IL の列を作る必要があった．この手順において取り出された ILGenerator は CreateType 呼び出しによる型生成が行われるまで常に有効であるため，他のメソッドの定義を行っているときなど，そのメソッドの定義とは関係のない部分で誤って用いることでメソッドの内容が意図せぬものになる場合がある．ILGenerator による IL 生成はアセンブリ言語によるプログラミングに近く，その内容の安全性を検証することは困難である．そこで本 API では，ILGenerator を直接操作せずにメソッド定義を行う手段を提供することにした．C#の標準ライブラリには式木（Expression Trees）と呼ばれる API が存在し，ラムダ式を構文木に変換する，構文木を自由に構築する，構築した構文木をメソッドとしてコンパイルして実行する，などの機能が備わっている．しかし式木 API は動的に生成する型のメソッドの定義には利用出来ない．本 API では，標準ライブラリの式木に類似した API を提供することで構文木によるメソッド本体の定義を実現した．ただし，本 API が提供する構文木クラスは全ての構文要素を網羅しているわけではないこと，そして構文木では表現不可能な IL を直接生成する必要のある場面が存在することから，既存 API と互換性のあるメソッドは残すこととした．

Add***メソッドのうち，AddProperty に関しては標準ライブラリの DefineProperty メソッドとは互換性を持たないバージョンが用意されている．DefineProperty メソッドは型にプロパティのみの定義を追加するメソッドであるため，プロパティにアクセサーを定義するためにはプロパティとは別にアクセサーとして用いるメソッドを定義し，さらに定義したメソッドをプロパティのアクセサーとして登録する必要がある．この手順は煩雑であり，バグの原因となりやすい．本 API の AddProperty はこの問題を解決するため，Get メソッドと Set メソッドの両方を持つプロパティを型に追加する AddProperty の他，Get メソッドのみを持つプロパティを型に追加する AddGetOnlyProperty と，Set メソッドのみを持つプロパティを型に追加する AddSetOnlyProperty を提供する．これらのメソッドは引数としてプロパティ名，プロパティの型，PropertyAttributes，Set メソッドと Get メソッドの MethodAttributes を取り，プロパティを表す PropertyBuilder とアクセサーを表す MethodBuilder の組を返す．これを用いることで，プロパティの定義とアクセサーの定義を同時に行うことが可能になる．

3.2.4　実行時型エラーとコンパイルエラー

C#は静的型付き言語であるため，全ての変数や式にはコンパイル時に静的な型が与えられている．動的に生成される型はコンパイル時には存在しないため，リフレクションによって型

情報を直接参照する場合を除き，動的な型とそのインスタンスは別の静的な型を介して利用することになる．標準ライブラリの API を用いた場合，Activator.CreateInstance メソッドによってインスタンスの生成を行うが，このメソッドの戻り値の型は object であるため，生成されたインスタンスは通常何らかのクラス型かインターフェース型にキャストしてから用いられる．

この手順において，実行時例外が発生するタイミングは CreateType メソッドによる型生成時と CreateInstance 後のキャスト時である．CreateType メソッド自身は生成中の型がどのような型にキャストして利用されるものか判断出来ないため，基底クラスの設定ミスやインタフェースの実装忘れは CreateType メソッドでは指摘出来ない．そもそも C#におけるダウンキャストは組み込み型など一部の自明なものを除き実行時にキャスト可能性が検査されるものであることを考えても，キャスト可能性をキャスト時に検査する仕様は適当であると言える．

動的に生成した型から生じる実行時エラーは，仮にその型に静的型検査を行った場合にもコンパイルエラーの原因になると考えられる．この考えを元に，我々は動的型生成に起因する実行時エラーをその誤りの内容とエラーの原因によって分類し，C#コンパイラのコンパイルエラーと対応付けた．以下に実行時エラーの原因となる誤りのうち典型的なものを列挙し，その詳細や関連するコンパイルエラーについても触れる．

- 型内部の識別子の重複
  C#は，メソッドのオーバーロードを除き，同じ型の中での識別子の重複を認めていない．一方，.NET は識別子の重複を認めているため，重複する識別子を持つ型を動的に生成した場合でも標準の CreateType メソッドはエラーを発生させない．しかし，そのような型のインスタンスのメンバーへのアクセスを C#から行った場合，AmbiguousMatchException 例外が発生する．この問題は本来，型の生成時に確認されるべきである．本 API では，型が重複する識別子を持つ場合，CreateType 実行時にエラーメッセージ内でその一覧を列挙する．C#における識別子の重複によるコンパイルエラーは，重複する識別子のカテゴリの組み合わせごとに用意されている．
- メンバーのアクセシビリティがその構成要素よりも高い
  メソッドのアクセシビリティがメソッドの戻り値の型よりも高い場合などを指す．このようなメソッドおよびその呼び出しを含むコードは動的に生成可能だが，メソッド呼び出しの際に MissingMethodException 例外が発生する．本 API では，このようなメソッドの定義は CreateType 実行時に検出される．C#における同種のコンパイルエラーは，戻り値の型に対するものとパラメータ型に対するものがそれぞれ用意されている．
- オーバーライドメソッドのシグネチャが元のメソッドと一致しない
  このエラーは標準ライブラリの CreateType メソッドでも発生するが，型生成時に検査が行われているわけではなく，生成された不正な型の読み込み時に発生した TypeLoadException 例外が報告されることによる．そのため，エラーメッセージは問題のあるメソッドの情報を含まない．本 API では CreateType 実行時にこの問題を検

出し，オーバーライドメソッドと元のメソッド双方の情報をエラーメッセージ内で報告する．C#のメソッドのオーバーライド指定は元のメソッドと同じシグネチャのメソッドに修飾子 override を付けることで行うが，メソッドのオーバーロードが認められている関係から，コンパイル時の同様のエラーは戻り値の型のみが異なる場合戻り値型の不一致，そうでない場合はオーバーライド元メソッドが存在しないことを原因とするエラーメッセージとして報告される．

- 必要なメンバーが存在しない
  動的型生成によるエラーの多くはこれに該当する．基底クラスやインタフェースが要求するメンバーを実装していない型を作った場合，CreateType 内の型の読み込み時に TypeLoadException として検出される．このエラーの発生理由は大きく分けると次のように分類出来る．
  - 名前の不一致
  - パラメータ型や戻り値型の不一致
  - MethodAttribute の不一致

  あるメソッドがインタフェースメソッドの実装となるためには，名前が完全に一致し，パラメータ型と戻り値型が ref 修飾を含めて一致し，さらに public かつ virtual である必要がある．これらの一つでも満たされていない場合，処理系はそのメソッドをインタフェースメソッドとは無関係なものと見なす．一方，同様の誤りが C#プログラムに含まれていた場合，コンパイラはそのミスを的確に指摘する．本 API は，インタフェースメソッドの実装が見つからなかった場合，型，MethodAttribute，名前が類似するメソッド定義の一覧をエラーメッセージ内で報告する．
- メソッドの本体がない
  abstract でないメソッドは本体を持たなければならない．このエラーは標準ライブラリの API でも CreateType 呼び出し時に検出される．

3.3　ライブラリの内部実装

3.3.1　型構築操作の情報の取得

本研究で提案するライブラリでは，実行時型エラーの原因となった型構築時の操作を型の生成時に指摘する．この機能を実現するためにはまず，実行時に型構築操作を記録する必要がある．

現在の C#には，C#5.0 および C#10.0 で追加された Caller Info という機能が備わっている．これは C++ における__FILE__マクロや__LINE__マクロに類似したもので，メソッドの呼び出し元情報をメソッドの引数として取得可能にする．Caller Info は CallerInfo 属性と呼ばれる以下の属性を用いて取得する．

- `CallerFilePath`

メソッドの呼び出し元のファイルパスを取得する

- CallerLineNumber
  メソッドの呼び出し元の行番号を取得する
- CallerMemberName
  メソッドの呼び出し元のメンバー名を取得する．呼び出し元がメソッドである場合はメソッド名，コンストラクタである場合はコンストラクタ名，プロパティである場合はプロパティ名が，イベントである場合はイベント名が取得出来るほか，ラムダ式や匿名メソッドなどの匿名関数から呼び出されている場合はコンパイル時生成された匿名関数の名前ではなくその外側のメソッドの名前が取得出来る
- CallerArgumentExpression
  メソッドの呼び出し元における，指定した引数に渡された実引数の文字列表現を取得する．引数を指定するため，この属性は引数の名前を表す文字列を引数にとる

プログラム 3.6 は，これらの属性を利用したメソッドを定義し利用する例である．

```
1       static void Main(string[] args)
2       {
3           MyAssertion.Assert(4 == Square(2));
4           MyAssertion.Assert(6 == Square(3));
5       }
6       static int Square(int x) => x * x;
7       public static class MyAssertion
8       {
9           public static void Assert(bool condition, [
                CallerArgumentExpression("condition")] string
                conditionExpression = null)
10          {
11              if (!condition) throw new ArgumentException($"
                    Condition \"{conditionExpression}\"is evaluated to
                     false.");
12          }
13      }
```

プログラム 3.6. Caller Info を利用したプログラムの例

このプログラムを実行すると以下のエラーが発生する．

```
1       Unhandled exception. System.ArgumentException: Condition "6 ==
            Square(3)"is evaluated to false.
2     at MyAssertion.Assert(Boolean condition, String
          conditionExpression) in Program.cs:line 79
3     at Program.Main(String[] args) in Program.cs:line 70
```

MyAssertion.Assert メソッドは bool 型の値と CallerArgumentExpression 属性の付いた string 型の既定値付きの値を引数に取る．Caller Info はこのように，CallerInfo 属性をメソッ

ドのオプション引数に付けることで取得できる．この例では引数 conditionExpression に属性が付けられているため，メソッド呼び出し時には"4 == Square(2)"や"6 == Square(3)"といった文字列表現がこの引数に渡される．MyAssertion.Assert メソッドはこのようにして得られた文字列表現を例外のメッセージに利用している．CallerInfo はこのようなデバッグ用途のほか，実行時のログを取る目的でも利用される．この機能を応用することで，型構築操作の API 呼び出しのログを取ることが可能である．

3.3.2　操作履歴の埋め込み

前述の手法で得られた型構築操作の情報は，エラー発生時に参照可能でなくてはならない．これを実現するためには，型構築操作の履歴をエラー発生時までどこかに保存しておく必要がある．この機能のシンプルな実現方法としてまず，型生成操作のログを記録する専用のクラスを用意し API 呼び出しの度にログを追記する方法が考えられる．この方法はさらに以下の 3 種類に分類出来る．

1. 静的クラスあるいはシングルトンインスタンスの単一のフィールドに全ての動的型の構築履歴を保存
2. 静的クラスあるいはシングルトンインスタンスに Dictionary を持たせ，それぞれの動的型の構築履歴を分けて保存
3. ログを記録するクラスのインスタンスを動的型ごとに生成し，それぞれのインスタンス内に構築履歴を保存

方法 1. は，エラーを発生させた型の情報を原因解析時にフィルタリングする必要があり，追加の計算コストがかかってしまうためよい設計とは言えない．

方法 2. は，実装が簡易かつ解析時のログの取得も容易である．実装に必要な Dictionary のキーは型に紐づいている値でなければならないが，.NET の型にそれぞれ一意な識別番号として割り当てられる GUID は CreateType 呼び出し前の型には存在しないため（GUID の取得は実行時例外となる），代わりに TypeInfo を用いる必要がある．しかし，CreateType 呼び出し前に TypeBuilder から得られる TypeInfo と CreateType 後に typeof 演算子によって取得した Type 型のインスタンスから得られる TypeInfo は同一性が判定出来ない（別の型と見なされる）ため，この方法のみではキャスト時のエラー原因の判定は不可能である．この問題は，CreateType の成功時，CreateType 前の TypeInfo と CreateType 後の TypeInfo を関連付ける別の Dictionary に TypeInfo の対応関係を保存することで解決出来る．

方法 3. は，TypeBuilder のインスタンス生成時に別の型のインスタンスを同時に生成し，生成した型の型情報からそのインスタンスにアクセスする手段を提供することで実現可能である．この手法の実現方法として，動的に生成する型に型構築操作の履歴を記録するクラスのインスタンスへの参照を保持する静的なフィールドを追加する方法がある．しかしこの方法を用いると生成する型の構造が暗黙のうちに変更されてしまうため，この方法は避けるべきである．

以上から，専用のロギングクラスを用いた方法で適当なものは方法 2. であると言える．

ところで，3.2.1 で触れたように C#には属性（Attributes）と呼ばれる機能が存在する．属性はコンパイラやエディタへの指示に利用されるほか，ライブラリへのマーカーとして多くの場面で利用されている．標準ライブラリ以外のものも含め，いくつかの例を以下に示す．

- **ConditionalAttribute**
  条件コンパイルに用いる．この属性がついたメソッドは特定のシンボルが定義されているときのみ有効になり，無効な場合はメソッド定義とメソッド呼び出しがコンパイル時に削除される
- **AttributeUsageAttribute**
  ユーザー定義属性の適用範囲の指定に用いる
- **DllImportAttribute**
  この属性がついたメソッドはネイティブな（.NET の管理化にない）DLL から読み込まれる
- **RequireComponentAttribute（Unity）**
  ゲーム開発ツールの Unity では，ゲーム内のオブジェクト（プレイヤーキャラクタや UI 要素などを指す）が多数のコンポーネントと呼ばれる要素の組み合わせによって構成される．それぞれのコンポーネントは MonoBehaviour クラスの派生クラスとして定義され，自身やその他のオブジェクトを構成するコンポーネントには GetComponent メソッドによりアクセスが可能となっている．これらのコンポーネントは依存関係を持つ場合がある．例えば，時間経過によって 3D モデルの色が変化するオブジェクトを作る場合，3D モデルのデータを保持するコンポーネント (Mesh) と保持された 3D モデルの描画を担うコンポーネント (MeshRenderer) に加え，MeshRenderer にアクセスし描画色を変更するための自作コンポーネント（名前は自由に決められるが以下では ColorChanger とする）が必要である．このようなとき ColorChanger に RequireComponent 属性を与えることで, その動作に MeshRenderer コンポーネントが必要であることを Unity に明示出来る．これにより，オブジェクトへの ColorChanger コンポーネント追加時に MeshRenderer コンポーネントが存在しない場合自動的にコンポーネントが追加されるようになるほか，ColorChanger コンポーネントが付いている限り MeshRenderer コンポーネントが外せなくなる．
- **JsonIgnoreAttribute**
  標準ライブラリの JsonSerializer が用いる属性である．JsonSerializer は標準で全てのプロパティをシリアライズ対象にするが，JsonIgnoreAttribute を付けたプロパティはその対象外となる．

動的に生成される型の構築操作の履歴は，その型に付随する情報に他ならない．エラーの発生時，エラーを発生させた型が型構築操作の履歴を属性として保持していれば，その履歴の解析は容易である．この方針は，先ほど挙げた方法 2. のようにロギング用のクラスを用意する必要がなく，より簡潔である．よって本 API の実装では，型構築操作の履歴を生成対象の型

自体に属性として埋め込み，エラーの発生時に属性情報からエラーの原因を特定する手法を採用した．

動的に生成される型への属性の追加は，標準ライブラリ API の SetCustomAttribute メソッドによって実現できる．本 API では 1 回の型構築操作を表す以下の属性を用意し，AddMethod などそれぞれの操作の内部でこれらの属性を SetCustomAttribute によって生成対象の型へ埋め込んだ．

- **DynamicTypeCreationHistoryAttribute**
  以下の全ての属性の基底クラスである属性．操作の種類，ファイルパス，行番号，操作を行ったメンバーの名前の情報からなる．
- **AddMethodHistoryAttribute**
  型へのメソッド追加操作の履歴を表す属性．メソッド名，戻り値の型およびパラメータ型の情報を持つ．
- **AddFieldHistoryAttribute**
  型へのフィールド追加操作の履歴を表す属性．フィールドの名前と型情報を持つ．
- **AddConstructorHistoryAttribute**
  型へのコンストラクタ追加操作の履歴を表す属性．パラメータ型の情報を持つ．
- **AddPropertyHistoryAttribute**
  型へのプロパティ追加操作の履歴を表す属性．プロパティ名，プロパティの型，Set メソッドおよび Get メソッドの情報を持つ．
- **AddIndexerHisrotyAttribute**
  型へのインデクサー追加操作の履歴を表す属性．インデクサーの型，添字の型，Set メソッドおよび Get メソッドの情報を持つ．
- **AddInterfaceImplementationHistoryAttribute**
  型へのインタフェースの実装の追加操作の履歴を表す属性．実装を追加したインタフェースの型情報を持つ．

3.3.3　エラーメッセージの生成

型やメンバーが持つ属性は，リフレクション API の GetCustomAttributes メソッドによって取り出すことが出来る．本 API ではこれを用いて，CreateType の実行時およびキャスト演算の実行時に生成した型に埋め込まれた操作履歴の情報を取得し内容を分析する．

CreateType での整合性チェックは，以下の順番で行われる．なお現在の実装では，整合性チェックの間に型エラーが検出された場合はそこでチェックを打ち切り，以降のチェックを行わない．

1. 型そのものが持つ情報の整合性チェック
2. それぞれのフィールドの整合性チェック

3. フィールド間の整合性チェック
4. それぞれのメソッドの整合性チェック
5. メソッド間の整合性チェック
6. それぞれのプロパティおよびインデクサーの整合性チェック
7. プロパティおよびインデクサー間の整合性チェック
8. メンバー間の整合性チェック
9. 実装したインタフェースとの整合性チェック
10. 基底クラスとの整合性チェック

型そのものが持つ情報とは型名やアクセシビリティ，カテゴリ（クラス，構造体，インタフェースのいずれであるか）および基底クラスの情報である．型名が既存の型と競合する場合やクラス以外の型が基底クラスを持つ場合，基底型が構造体である場合，基底クラスよりも高いアクセシビリティを持つ場合および基底クラスが不完全な型の場合はこの段階で型エラーが検出される．

フィールドの整合性チェックでは，フィールドがその型よりも高いアクセシビリティを持つ場合やインタフェース型がインスタンスフィールドを持つ場合，そしてフィールドの型が不完全な場合に型エラーが検出される．またフィールド間では識別子の競合がチェックされ，競合を検出した場合は競合を起こした同名フィールドの追加操作をリスト化する．

メソッドの整合性チェックでは，以下の項目が検証される．

- アクセシビリティ
  メソッドがパラメータ型や戻り値の型よりも高いアクセシビリティを持つ場合型エラーとなる．
- 抽象性
  abstract メソッドの本体が定義されている場合，非 abstract メソッドが本体を持たない場合，非 abstract クラスや構造体が abstract メソッドを持つ場合は型エラーとなる．
- メソッド間の識別子の重複
  メソッド名が同一であり，かつパラメータ型リストが一致する複数のメソッドが見つかった場合型エラーとなる．このような場合，重複するメソッド定義の追加操作を実行時の履歴から取り出しリスト化する．ただし，C#および.NET はメソッドのオーバーロードを許容しているため，メソッド名が同一であってもパラメータ型が異なる場合は型エラーとならない．
- 不正なオーバーライド
  1 つのメソッドが複数のメソッドによってオーバーライドされている場合および基底クラスのものではないメソッドをオーバーライドするメソッドを定義した場合，このチェックの段階で型エラーとなる．C#におけるメソッドのオーバーライドは基底クラスのメソッドと同名のメソッドに修飾子 override を付けることで行われるため，この型エラーには対応するコンパイルエラーが存在しない．

プロパティおよびインデクサーの整合性チェックでは，以下の項目が検証される．なお，C#のインデクサーは特殊な名前のメソッド get_Item および set_Item からなるプロパティであるため，ここでは両者を同じものとして扱う．

- アクセサー
  プロパティが Get メソッドも Set メソッドも持っていない場合，型エラーとなる．
- アクセシビリティ
  プロパティのアクセサー（Get メソッドおよび Set メソッド）がプロパティよりも高いアクセシビリティを持つ場合やプロパティがその型よりも高いアクセシビリティを持つ場合，型エラーとなる．
- 抽象性
  abstract なプロパティが非 abstract なアクセサーを持っている場合および非 abstract なプロパティが abstract なアクセサーを持っている場合，型エラーとなる．
- アクセサーの型
  Get メソッドの戻り値の型とアクセサーの型が一致しない場合や，Set メソッドの引数の型がアクセサーの型と一致しない場合に型エラーとなる．なお，これらの型は C#では明示的に表されないため，この実行時エラーに対応するコンパイルエラーは存在しない．インデクサーの場合はこれに加えてインデクサーの引数の型に対しても同様のチェックを行う．
- プロパティ間の識別子の重複
  同名のプロパティが複数定義されている場合型エラーとなる．この場合，重複するプロパティ定義の追加操作を実行時の履歴から取り出しリスト化する．

メンバー間の整合性チェックでは，以下の検証を行う．

- 異なるカテゴリのメンバー間の識別子の重複
  C#はメソッドのオーバーロードを認めているが，同名のフィールドとメソッド，同名のメソッドとプロパティなど，異なるカテゴリ間の識別子の重複は認めていない．よってこれらは型エラーとなる．この場合，識別子が重複する全てのメンバー追加操作を実行時の履歴から取り出しリスト化する．

実装したインタフェースとの整合性チェックでは，以下の検証を行う．

- メソッド実装の網羅性
  インタフェースの全てのメソッドは，インタフェースを実装する全ての型で実装されなければならない．非 abstract なクラスまたは構造体がデフォルト実装を持たないメソッドの実装を欠いている場合，型エラーとなる．.NET ランタイムは，あるメソッドがインタフェースのメソッドの実装であるか否かをメソッド名，戻り値を含むシグネチャ，そしてメソッドのアクセシビリティから判断する．本 API ではメソッド実装が見つからなかった場合，C#コンパイラのコンパイルエラーにおけるエラー原因の分類

に従い，その原因を以下の手順で分析する．

1. 非 public かつメソッド名とパラメータ型が一致するメソッドを探す
   実装メソッドのアクセシビリティ設定に問題があると判断する．コンパイルエラー CS0737 に相当．
2. public かつメソッド名とパラメータ型が一致し，戻り値型の ref 修飾が異なるメソッドを探す（戻り値型自体が異なる場合も含む）
   実装メソッドの戻り値の修飾に問題があると判断する．コンパイルエラー CS8152 に相当．
3. public かつメソッド名とパラメータ型が一致し，戻り値型が異なるメソッドを探す
   実装メソッドの戻り値に問題があると判断する．コンパイルエラー CS0738 に相当．
4. メソッド名が一致するメソッドを探す
   メソッドのパラメータ型に問題があると判断する．C#がメソッドのオーバーロードをサポートしていることからコンパイラはパラメータ型の異なるメソッドの存在をコンパイルエラーメッセージで指摘しないため，このエラーに相当するコンパイルエラーは存在しない．
5. 類似する名前を持つメソッドを探す
   タイプミスあるいは文字列結合のミスが原因であると判断する．ここでは，名前の類似度を 2 つのメソッド名のダメラウ・レーベンシュタイン距離を長いほうの名前で割って正規化することによって求める．このエラーにも相当するコンパイルエラーは存在しない．
6. 該当するメソッド定義が欠落していると判断する

- プロパティ実装の網羅性
  メソッドと同様，インタフェースの全てのプロパティはそれを実装する全ての型で実装されなければならない．非 abstract なクラスまたは構造体がデフォルト実装を持たないプロパティの実装を欠いている場合，型エラーとなる．プロパティ実装の網羅性チェックは以下の手順で行われる．
  1. 非 public かつプロパティ名が一致するプロパティを探す
     プロパティのアクセシビリティ設定に問題があると判断する．コンパイルエラー CS0737 に相当．
  2. public かつプロパティ名が一致し，型の ref 修飾が異なるプロパティを探す
     プロパティ型の修飾に問題があると判断する．コンパイルエラー CS8152 に相当．
  3. public かつプロパティ名が一致し，型の異なるプロパティを探す
     プロパティの型に問題があると判断する．コンパイルエラー CS0738 に相当．
  4. public かつプロパティ名が一致し，Get メソッドか Set メソッドのどちらかが public でないプロパティを探す
     アクセサーのアクセシビリティ設定に問題があると判断する．コンパイルエラー CS0277 に相当．

5. public かつプロパティ名が一致し，インタフェースのプロパティが要求するアクセサーを持たないプロパティを探す
   アクセサー定義が存在しないことが原因であると判断する．CS0535 に相当.
6. 類似する名前を持つプロパティを探す
   タイプミスや文字列連結のミスが原因であると判断する．手法はメソッドの場合と同様.

基底クラスとの整合性チェックでは，以下の検証を行う.

- 基底クラスが持つ abstract メソッドの実装定義の網羅性
  非 abstract クラスは abstract メソッドを持つことが出来ないため，非 abstract クラスが基底クラスの abstract メソッドに実装の定義を与えていない場合は型エラーとなる．本 API ではこのエラーを検出したとき，インタフェースの実装メソッドが見つからなかった場合と同様の手順によってその原因を分析する.
- override 指定されたメソッドの妥当性
  派生クラスのメソッドによってオーバーライド可能なのは，基底クラスにおいて virtual または sealed 指定されたメソッドに限られる．基底クラスの virtual でないメソッドまたは sealed 指定されたメソッドをオーバーライドするメソッドを定義した場合，型エラーとなる．また，オーバーライド元のメソッドとオーバーライドメソッドの型が一致しない場合にもエラーとなる.

これらの整合性チェックで問題が検出された場合，IncompleteTypeDefinitionException 例外が発生する．この例外が出力するエラーメッセージはエラーの内容を表しており，その原因となった操作と関連するコンパイルエラーの ID を含むようになっている．例えば，同名かつ同シグネチャのメソッド Function() を 2 つ定義した型を生成しようとした場合，以下のようなエラーメッセージが出力される.

```
1 Unhandled exception. IncompleteTypeDefinitionException: Duplicated
      method definition: Function() @
2         TestMain() in TestProgram.cs:line 38,
3         TestMain() in TestProgram.cs:line 38   [RELATED ERROR:
              CS0111]
4    at Program.Main(String[] args) in Program.cs:line 18
```

動的に生成される型が含む識別子はプログラム内においては文字列であるため，エディタによる自動補完が可能な通常のソースコード以上にタイプミスのリスクがある上，実行時の文字列結合によって作られる識別子はコンパイル時に確認出来ない．これを考慮し，本 API の整合性チェックは必要なメンバーが見つからなかった場合に操作履歴の中から必要なメンバーに近い識別子を探すよう設計されている．識別子の類似性は文字列同士の距離を元に計算している.

適切なエラーメッセージを生成するため，我々は C#のコンパイラが出力するエラーメッ

セージのうちリファレンスで確認可能なものを全て精査し，そのうち型エラーに関連するものを動的型エラーと対応付けた．この対応表は付録に記す．

キャスト演算によって発生する実行時エラーは，生成した型とキャスト先の型に互換性がない場合に生じる．このエラーの発生理由は次の 3 つに分類出来る．

1. キャスト先の型の指定を誤っている
2. キャスト先のインタフェース型を実装していないか生成した型に基底クラスが存在しない
3. キャスト先のクラスと互換性のない基底クラスから派生している

1. の場合は実行時エラーの発生箇所を確認することで問題が解決可能であり，またこのケースは動的型生成による実行時エラーとは言えないことから本研究の対象ではない．2. の場合は，AddInterfaceImplementation メソッドの呼び出しや基底クラスの設定が型構築操作に欠けているはずである．よってこの場合に生成されるエラーメッセージはこれらの操作を CreateType より前に実行するように促すものが適当であると考えられるが，これだけでは不十分である．なぜなら，インタフェース実装や基底クラスの指定忘れがエラーの原因である場合，これらを指定することによって新たなエラーを生じる可能性があるためである．よって本 API ではこのようなケースにおいて，追加すべき操作が行われていた場合 CreateType メソッド内の型検査で検出されると考えられるエラーについても検証し，キャスト時のエラーメッセージにその情報を含めることとした．3. の場合は，動的型生成時に基底クラスを指定した操作の場所を探し，その位置の情報をエラーメッセージに含める．また，キャスト先のクラスが基底クラスに指定されていた場合に発生すると考えられるエラーについても，2. と同様に検査を行う．なお，1. のケースは 2. および 3. のケースと区別することができないため，場合によって 2. または 3. のどちらかのプロセスでエラーメッセージが生成される．

なお本 API が提供する型変換は，厳密にはキャスト演算子ではなくメソッドである．C#では，キャスト演算子の動作を変更したい場合にはその対象の型ごとにキャスト演算子のオーバーロードを定義する必要がある．そのため，任意の型に対するキャスト演算子の動作をあらかじめ変更することは出来ない．本 API でキャスト演算子の代わりに提供するメソッドは，その内部で独自にキャストの安全性を確認し，型変換可能であると判断した場合に通常のキャスト演算を実行するようになっている．

第 4 章

実験

本研究で作成したライブラリを用いたプログラムのソースコードに意図的にバグを混入させることでエラーを起こし，そのエラーメッセージを分析する実験を行った．混入したバグの原因ごとに望ましいエラーメッセージを考え，標準ライブラリを用いた場合および本研究のライブラリを用いた場合のエラーメッセージと比較する．ソースコードへのバグの意図的な混入は，後述するプログラムを用いたソースコード変換によって行う．変換されたプログラムは変数定義の消滅など本研究の対象とは直接関係のない部分が原因となるコンパイルエラーを生じさせる場合があるため，本実験では変換後にコンパイル可能であったもののみを分析対象とする．実験対象として，C#で実装された高速な JSON シリアライザである Utf8Json の内部実装を本研究のライブラリで置換したものと，簡素な構文を持つプログラミング言語の処理系を用いる．これらの詳細については以降の節で詳しく触れる．実験環境は Windows 10 Pro(21H1)，Intel(R) Core i9-9900K CPU @ 3.60GHz × 8，メモリ 64GB のマシンの Windows Subsystem for Linux 上の Ubuntu 20.04 である．

4.1 Utf8Json への適用

Utf8Json はシリアライズ対象の型に応じて最適化されたシリアライザを実行時に動的生成することで高速化を実現した，C#用の JSON シリアライザである．通常，オブジェクトから文字列フォーマットである JSON への変換はキーと値それぞれのシリアライズ結果を結合することで行われている．一方，UTF8Json はキー（C#の型のプロパティ名）の文字列の長さが型ごとに固定の値であることに着目し，その長さの byte 列のコピーを行うようにコードを生成することでシリアライズ時の分岐を削減している．また，数値や文字列値についても UTF8 へのエンコードを避け byte 列に直接書き込みを行うことでオーバーヘッドを回避している．そして JSON からオブジェクトへの変換では，変換対象の型のプロパティの一覧を表すトライ木を生成し，それを用いたキー文字列のマッチングを入力 byte 列に対して直接行うことで高速化を実現している．これらの機能の実現には標準ライブラリの動的型生成 API が用いられている．

本実験ではまず，Utf8Json が用いている動的型生成 API を我々の開発した API で置き換

えたものを作成した．次にプログラム 4.1 を実行し，ライブラリの動作確認を行った．

```
using System.Collections.Generic;
using Utf8Json;
public struct Item
{
    public int x { get; set; }
    public (int, int) y { get; set; }
    public override string ToString() => $"x: {x}, y: {y}";
}
public class User
{
    public string name { get; set; }
    public List<Item> items { init; get; } = new();
}
public class Program
{
    public static void Main(string[] args)
    {
        var deserialized = JsonSerializer.Deserialize<Item>(@"{""x
            "":100,""y"":{""Item1"":200,""Item2"":392}}");
        System.Console.WriteLine(deserialized);
        var serializedBytes = JsonSerializer.Serialize(new User
        {
            name = "Alice",
            items = {
                new Item{
                    x = 10,
                    y = (20, 30)
                },
                new Item
                {
                    x = 25,
                    y = (37, 11)
                }
            }
        });
        System.Console.WriteLine(JsonSerializer.PrettyPrint(
            serializedBytes));
    }
}
```

プログラム 4.1. Utf8Json を用いたプログラム

このプログラムを実行すると以下の出力が得られる．

```
x: 100, y: (200, 392)
{
  "name": "Alice",
  "items": [
    {
      "x": 10,
      "y": {
        "Item1": 20,
        "Item2": 30
      }
    },
    {
      "x": 25,
      "y": {
        "Item1": 37,
        "Item2": 11
      }
    }
  ]
}
```

JSON 形式によるシリアライズとデシリアライズがユーザー定義型に対して正常に動作していることが確認できる．

続いて，このプログラムから標準出力を行う部分を取り除いたものを正規の Utf8Json および改造した Utf8Json の双方について 1300 回ずつ実行し，動作のパフォーマンスを調べた．この実験の結果は以下のようになった．改造した Utf8Json は動作速度が約 1 割低下してい

表 4.1. 動作速度の比較

	正規の Utf8Json	改造した Utf8Json
実行時間 (s)	0.119	0.130

る．パフォーマンスを重視して設計された Utf8Json において，1 割の性能低下は無視できるものではない．しかし，本ライブラリを用いることによって追加のオーバーヘッドが発生するのはある型に対して初めてシリアライザやデシリアライザが呼び出された場合の型生成時のみでありシリアライズおよびデシリアライズ操作自体の速度には影響しないため，多くの場合このパフォーマンスの低下は問題にならないと考えている．また，パフォーマンス低下が無視できない場合，この問題はリリースビルド時に本ライブラリのチェック機構を除去することで事実上解決する．

次に，改造された Utf8Json のソースコードのうち型生成に関わる部分について，以下のソースコード変換操作をランダムに行った．

- 文字列リテラルの完全な置き換え
- 文字列リテラルに対する文字置換，前後の文字の入れ換え，文字挿入，文字削除
- 型構築操作の削除
- メンバーの Attributes の変更

その後，バグが混入した UTF8Json をプログラム 4.1 から利用し，実行時エラーメッセージを確認した．なお，ソースコード変換には Roslyn API を用いた自作のプログラムを使用した．

実験で混入したバグの内容と正規の Utf8Json による実行時エラーメッセージ，改造した Utf8Json による実行時エラーメッセージおよび対応するコンパイルエラーメッセージの対応を表 4.2 に示す．ただし，どちらのエラーメッセージも型やメソッドの完全名，ファイルのフルパスを含んでおり文字数が多いため，適宜省略する．

正規の Utf8Json が出力する実行時エラーメッセージは，問題のあるメソッドの名前を含んでいるものの，型の読み込み時にメソッドの実装が見つからなかったことのみを表しており，バグの内容によらずほとんど同じ文章である．一方，開発したライブラリを組み込んだ Utf8Json が出力する実行時エラーメッセージは，発生した問題の詳細，関連すると思われる他のメンバー定義が行われた場所，そして関連するコンパイルエラーの ID を含んでおり，バグの修正に活用できる．ただし，3 つめの例のようにメンバーのアクセシビリティを本来より制限の緩いものにする変更は，その操作によってオーバーライド元メソッドなどとのシグネチャの不一致が発生する場合やメソッドが引数および戻り値の型より高いアクセシビリティを持つことになってしまう場合を除き，問題として検出することが出来ない．また，生成対象の型が実装するインタフェースの型パラメータ指定を消去した 4 つめの例では，型パラメータが存在しないことではなくメソッドの引数の型が一致しないことがエラーの原因として報告されてしまっている．これは，開発したライブラリがクラスの実装するジェネリックインタフェースや基底クラスの型パラメータが指定されているかどうかを確認する仕様になっていなかったために発生した問題である．この挙動はライブラリの実装漏れによるバグであるため，実験後にライブラリの当該箇所を修正し，この問題に対応した．現在，同様のバグが混入した場合以下のエラーメッセージが表示される．

```
1 IncompleteTypeDefinitionException: Generic interface type 'Utf8Json.
      IJsonFormatter`1[T]' is unbound. Implemented at DefineType() in
      /Utf8Json/src/Utf8Json/Internal/Emit/DynamicAssembly.cs:line 42
        [RELATED ERROR: CS7003]
```

4.2 内部 DSL の作成

本研究で開発したライブラリを用いて，簡素な機能を持つ内部 DSL を作成した．この言語は以下の機能を持つ．

表 4.2. バグと実行時エラーメッセージの対応

バグの内容	正規の Utf8Json	改造した Utf8Json
Serialize メソッドの第一引数の型を ref JsonWriter から ref JsonReader へ変更する	System.TypeLoadException: Method 'Serialize' in type 'Utf8Json.Formatters.ItemFormatter1' from assembly ' 略' does not have an implementation.	IncompleteTypeDefinitionException: Method 'Void Serialize(略)' required by interface ' 略' is missing. Methods with same name: System.Void Serialize(略) @ BuildType() in DynamicObjectResolver.cs:line 581 [RELATED ERROR: CS0535]
Deserialize メソッドのアクセシビリティを public から private に変更する	System.TypeLoadException: Method 'Deserialize' in type 'Utf8Json.Formatters.ItemFormatter1' from assembly ' 略' does not have an implementation.	IncompleteTypeDefinitionException: Method 'Item Deserialize(略)' required by interface Utf8Json.IJsonFormatter‘1[Item] must be public. Defined at BuildType() in DynamicObjectResolver.cs:line 601' [RELATED ERROR: CS0737]
フィールド stringByteKeys のアクセシビリティを private から public へ変更する	エラーなし	エラーなし
Formatter クラスが実装するインタフェースの型から型パラメータ情報を消す	System.TypeLoadException: Method 'Serialize' in type 'Utf8Json.Formatters.ItemFormatter1' from assembly ' 略' does not have an implementation.	IncompleteTypeDefinitionException: Method 'Void Serialize(JsonWriter ByRef, T, IJsonFormatterResolver)' required by interface 'IJsonFormatter‘1[T]' is missing. Methods with same name: Void Serialize(JsonWriter& , Item, IJsonFormatterResolver) @ BuildType() in DynamicObjectResolver.cs:line 581 [RELATED ERROR: CS0535]

- コマンド定義
- Define コマンドによるローカル変数定義
- 変数への代入と四則演算付き代入
- 変数の中身や数値の表示
- ループ
- コマンド呼び出し

コマンドは命令の列として定義され，命令はコマンド名に任意の数の引数を付けたものである．サポートしている型は int 型と string 型のみであり，ユーザー定義型は存在しない．この DSL を用いたプログラムの例をプログラム 4.2 に示す．

```
1  public interface IRunner
2  {
3      void Run(params string[] args);
4  }
5  static class Program
6  {
7      public static void Main(string[] args)
8      {
9          var engineFactory = new EngineFactory();
10         // define new recipe: void Run(string[] args){int x=20; x
               +=10; int y=2; SubProcess(x,y);}
11         var runtimeEngine_t = engineFactory
12         .AddRecipe(
13             new("Run", true, typeof(void), new[] { (typeof(string[])
                   , "args") }
14             , new("Define", "int", "x")
15             , new("Assign", "x", "20")
16             , new("Add", "x", "10")
17             , new("Define", "int", "y")
18             , new("Assign", "y", "2")
19             , new("SubProcess", "x", "y")
20         ))
21         // define new recipe: void SubProcess(int left,int right){
               int answer = left;for(int i=0;i<10;i++){answer*=right;
               Show(answer);}}
22         .AddRecipe(
23             new("SubProcess", false, typeof(void), new[] { (typeof(
                   int), "left"), (typeof(int), "right") }
24             , new("Define", "int", "answer")
25             , new("Assign", "answer", "left")
26             , new("Repeat", "i", "0", "10")
27             , new("Mul", "answer", "right")
```

```
28              , new("Show", "answer")
29              , new("EndRepeat", "i")
30          ))
31          .Create();
32          var instance = runtimeEngine_t.CreateInstanceAs<IRunner>();
33          instance.Run();
34      }
35  }
```

プログラム 4.2. DSL を利用したプログラム

このプログラムは，メソッド Run とメソッド SubProcess を持ちインタフェース IRunner を実装するクラスを実行時に生成し，そのインスタンスの Run メソッドを実行する．生成された Run メソッドはその内部で SubProcess メソッドを呼び出す．このプログラムによって生成されるクラスに相当する C#のコードを以下のプログラム 4.3 に示す．

```
1  public class RuntimeEngine : IRunner
2  {
3      public void Run(string[] args)
4      {
5          int x = 20;
6          x += 10;
7          int y = 2;
8          SubProcess(x, y);
9      }
10     public void SubProcess(int left, int right)
11     {
12         int answer = left;
13         for (int i = 0; i < 10; i++)
14         {
15             answer *= right;
16             Console.WriteLine(answer);
17         }
18     }
19 }
```

プログラム 4.3. 動的に生成されるクラスの C#表現

次に，同様に動作する DSL を標準ライブラリの API のみを用いて作成した．そして，DSL 内にバグを混入させ，発生する実行時エラーとそのエラーメッセージを調べた．

はじめに，この DSL およびプログラム 4.2 を作成する過程で発生した実行時エラーメッセージとその解決に必要であった手順を表 4.3 に，標準ライブラリの API のみによって実装した DSL に対して同様のバグを意図的に混入させた際に発生したエラーメッセージを表 4.4 に示す．標準ライブラリによるエラーメッセージはその原因に関わらず同じ内容であり，エラー原因の特定に有用であるとは言い難い．一方，開発したライブラリによるエラーメッセー

ジはその原因に応じて内容が異なっており，問題解決のヒントとなる情報が含まれている．

表 4.3. DSL 作成中に生じた実行時エラー（開発したライブラリ）

実行時エラーメッセージ	解決策
Method 'Void Run(System.String[])' required by interface 'IRunner' is missing. Methods with similar name: run @AddRecipe() in Program.cs:line 51 [RELATED ERROR: CS0535]	run メソッドの名前を Run に変更する
Method 'Void Run(System.String[])' required by interface 'IRunner' is missing. Methods with same name: System.Void Run(System.String) @ AddRecipe() in Program.cs:line 51 [RELATED ERROR: CS0535]	Run メソッドの引数の型を `string` から `string[]` に変更する
Method 'Void Run(System.String[])' required by interface IRunner must be marked virtual. Defined at AddRecipe() in Program.cs:line 51'	`MethodAttributes.Virtual` をメソッドの性質に追加する

続いて，これらとは異なる誤りをプログラムに混入させた場合の結果を以下の表 4.5 に示す．

1 つめのバグによるエラーは，いずれも Run メソッドの名前が un になったことにより動的に生成された型がインタフェース IRunner の Run メソッドを実装しなくなったために発生している．このようなバグは，メンバー名のタイプミスや文字列操作のミスによって発生する場合がある．開発したライブラリは似た名前である un メソッドが型に追加された場所をエラーメッセージ内に含めており，開発者はミスに気づく可能性が高い．2 つめのバグによるエラーは，今回開発したライブラリと DSL では捕捉することが出来なかった．このエラーは，実行時に生成された SubProcess メソッドの実行終了時，SubProcess メソッドの戻り値の型が void ではないため処理系が Ret 命令によりスタックに残った値を取り出そうとしたところ，スタックが空であったことが原因で発生している．本研究で開発したライブラリは型構築操作の履歴を用いて型の異常とその原因を検知するが，メソッド本体の処理の整合性を検証する機能は持ち合わせていないため，この種のエラーには対応出来ない．本研究で開発したライブラリは式木の導入によりこの問題の発生リスクを抑える設計になっているが，Utf8Json がプロパティ名をトライ木に格納し利用していたように動的型生成には C#や式木では表現することの出来ない処理を行う必要のある場面が存在することから，この問題の解決には本研究とは別のアプローチが必要である．

表 4.4. DSL 作成中に生じた実行時エラー（標準ライブラリ）

実行時エラーメッセージ	解決策
Method 'Run' in type 'RuntimeEngine' from assembly 'Dynamic, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null' does not have an implementation.	run メソッドの名前を Run に変更する
Method 'Run' in type 'RuntimeEngine' from assembly 'Dynamic, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null' does not have an implementation.	Run メソッドの引数の型を `string` から `string[]` に変更する
Method 'Run' in type 'RuntimeEngine' from assembly 'Dynamic, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null' does not have an implementation.	`MethodAttributes.Virtual` をメソッドの性質に追加する

表 4.5. DSL を用いたプログラムに混入させたバグとエラーメッセージ

バグの内容	標準ライブラリ	開発したライブラリ
コマンド名の最初の文字が削除される	System.TypeLoadException: Method 'Run' in type 'RuntimeEngine' from assembly 'Dynamic' does not have an implementation.	IncompleteTypeDefinitionException: Method 'Void Run(System.String[])' required by interface 'IRunner' is missing. Methods with similar name: un @AddRecipe() in Program.cs:line 55 [RELATED ERROR: CS0535]
SubProcess メソッドの戻り値の型を void から int にする	System.InvalidProgramException: The JIT compiler encountered invalid IL code or an internal limitation. at RuntimeEngine.Run(String[] args)	

第 5 章

まとめと今後の課題

5.1 まとめ

本研究では動的な型構築操作の履歴を生成対象の型自身にメタデータとして組み込むことで実行時型エラーの発生時にその原因を分析可能にする仕組みを提案し，実験によって実行時型エラーメッセージがより詳細なものになることを示した．動的に生成された型が引き起こすエラーの多くはコンパイル時の型検査が存在しないことに由来しており，そのエラーメッセージからは処理系が型の異常を検出したことしか分からない場合が多い．また，異常を含む型の読み込みが成功してしまうケースもあり，そのような場合バグの発見はメンバーへのアクセス時まで遅延してしまう．これらの問題は，C#が持つ静的型付き言語の利点を損なっている．

型構築時に操作の履歴を保存し型生成時にそれを用いた型検査を実施することで，不正な型の生成を抑制できるだけでなく，不正な型が生成された場合にその原因を特定することが容易になる．

型構築時に埋め込まれた操作履歴を用いて型検査を行う機能を備えた動的型生成ライブラリを開発し，高機能なライブラリや動的な処理を行うアプリケーションへ組み込む実験によってエラーメッセージの具体性が向上することを確認した．

5.2 今後の課題

メタプログラミングの普及には，適切なエラーメッセージと網羅的かつ可能な限り早い段階でのエラー検出による開発の容易化が必要不可欠である．本研究で開発したライブラリは型が含む誤りを型の生成時に検査するが，可能であれば静的な解析によってコンパイル時にエラーを検出するほうが望ましい．API の提供方法を工夫することによって，ソースジェネレータや T4 を活用したコンパイル時の整合性チェック機構を作ることが可能であると考えている．しかし，これらの仕組みには対象となる言語の仕様に大きく依存してしまうという欠点がある．Java は C#と同様バイトコードに変換され仮想マシン上で動作するように設計されている言語であり，C#の attribute に類似したアノテーションという機能を有する．また，スタックトレースの情報を用いて実行時のメソッド呼び出し位置を追跡可能であること，実行中にクラス

をロードする機能を有していることから，本研究のライブラリを Java に移植することは可能であると考えられる．しかし多くの言語は実行時に型を生成する手段を提供していないため，同じ手法を他の言語に拡張を加えず応用することは難しい．

一方，動的コード生成に限らず，エラーの発生箇所とその原因となった操作が離れているためにエラー原因の特定が容易でない場面は多くの言語に存在する．これらについては，エラーの発生原因となりうる操作の履歴を記録し分析することでエラーメッセージを改善する本研究の手法が有効である．

発表文献と研究活動

(1) 横井駿平，千葉滋．静的型付き言語で動的に生成された型が含むバグの早期発見に向けた研究．日本ソフトウェア科学会 第 38 回大会, 2021.09.01.

参考文献

[1] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pp. 23–35, New York, NY, USA, 1984. Association for Computing Machinery.

[2] Alan C. Kay. The early history of smalltalk. *SIGPLAN Not.*, Vol. 28, No. 3, pp. 69–95, mar 1993.

[3] Oleg Kiselyov. The design and implementation of BER metaocaml - system description. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, Vol. 8475 of *Lecture Notes in Computer Science*, pp. 86–102. Springer, 2014.

[4] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, Vol. 248, No. 1, pp. 211–242, 2000. PEPM'97.

[5] Walid Taha. *A Gentle Introduction to Multi-stage Programming*, pp. 30–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[6] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pp. 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery.

[7] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pp. 1–16, New York, NY, USA, 2002. Association for Computing Machinery.

[8] Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. Staging with class: A specification for typed template haskell. *Proc. ACM Program. Lang.*, Vol. 6, No. POPL, jan 2022.

[9] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in c++. *SIGPLAN Not.*, Vol. 41, No. 10, pp. 291–310, October 2006.

[10] Bjarne Stroustrup and Gabriel Reis. Concepts – syntax and composition. 10 2003.

[11] Brianna M. Ren and Jeffrey S. Foster. Just-in-time static type checking for dynamic languages. *SIGPLAN Not.*, Vol. 51, No. 6, pp. 462–476, jun 2016.

[12] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pp. 151–161, New York, NY, USA, 1986. Association for Computing Machinery.

[13] N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. Revised¡sup¿5¡/sup¿ report on the algorithmic language scheme. *SIGPLAN Not.*, Vol. 33, No. 9, pp. 26–76, sep 1998.

謝辞

本研究を進めるにあたり，適切な指導をしてくださった千葉滋教授，山崎徹郎特任助教に心より感謝いたします．研究の方向性が定まらないときには的確なアドバイスを頂き，論文執筆においても多くの助言を頂きました．定期的な進捗確認がなければ，私の研究は行き詰っていたと思います．2 年間ほとんど家から出なかった私の生活を支えていただいた父と母，海外から応援してくれた妹のおかげで私の研究生活は豊かなものになりました．また，東京工業大学博士後期課程の相澤遥也さんには友人として，そして隣接分野の研究者として様々な相談に乗っていただきました．本当にありがとうございました．

付録 A

コンパイル時型エラーと実行時型エラーの対応表

表 A.1: コンパイルエラーと実行時エラーの対応

エラーコード	エラーの内容	発生する実行時エラー
CS0050	メソッドの戻り値の型のアクセシビリティがメソッドより低い	エラーなし
CS0051	メソッドのパラメータ型のアクセシビリティがメソッドより低い	エラーなし
CS0052	フィールドの型のアクセシビリティがフィールドより低い	エラーなし
CS0053	プロパティの型のアクセシビリティがプロパティより低い	エラーなし
CS0054	インデクサーの戻り値の型のアクセシビリティがインデクサーより低い	エラーなし
CS0055	パラメータ型のアクセシビリティがインデクサーより低い	エラーなし
CS0056	戻り値の型のアクセシビリティが演算子より低い	エラーなし
CS0057	パラメータ型のアクセシビリティが演算子より低い	エラーなし
CS0058	戻り値の型のアクセシビリティがデリゲートより低い	エラーなし
CS0059	パラメータ型のアクセシビリティがデリゲートより低い	エラーなし

CS0060	基底クラスのアクセシビリティが派生クラスより低い	System.TypeLoadException: Access is denied
CS0061	基本インタフェースのアクセシビリティが派生インタフェースより低い	System.TypeLoadException: Type ' 型名' is attempting to implement an inaccessible interface.
CS0065	イベントが add アクセサーか remove アクセサーを持たない	エラーなし
CS0066	イベントの型がデリゲート型でない	System.InvalidOperationException: Cannot add the event handler since no public add method exists for the event.(add アクセサー呼び出し時)
CS0069	インタフェースのイベントのアクセサーが定義されている	エラーなし
CS0082	メンバーの識別子が自動生成される名前 (プロパティのアクセサーなど) と競合する	エラーなし
CS0100	メソッドのパラメータ名が重複する	エラーなし
CS0102	同一の型が重複する識別子を宣言している	AmbiguousMatchException: Ambiguous match found.(当該要素へのアクセス時)
CS0111	同じ名前とパラメータ型を持つメソッドが複数存在する	AmbiguousMatchException: Ambiguous match found.(当該要素へのアクセス時)
CS0112	static メソッドに virtual や abstract, override を指定している	System.ArgumentException: Method cannot be both static and virtual.
CS0112	オーバーライド元のメソッドが見つからない	明示的に指定する仕様のため発生しない
CS0132	静的コンストラクターがパラメータを持つ	System.TypeLoadException: Could not load type ' 型名'
CS0146	クラスの継承が循環を含む	System.TypeLoadException: Could not load type ' 型名'
CS0179	extern メソッドの本体が存在する	System.InvalidOperationException: Method body should not exist.
CS0180	メソッドが extern かつ abstract である	System.TypeLoadException: Could not load type ' 型名'

CS0215	true 演算子または false 演算子の戻り値の型が bool でない	エラーなし (演算子のオーバーロードと見做されない)
CS0216	==演算子と!=演算子の片方が定義されていない または true 演算子と false 演算子の片方が定義されていない	エラーなし
CS0239	sealed メソッドをオーバーライドしている	System.TypeLoadException: Declaration referenced in a method implementation cannot be a final method.
CS0273	アクセサーのアクセシビリティがプロパティよりも高い	仕様上発生しない
CS0275	インタフェースのアクセサーのアクセシビリティが public でない	エラーなし
CS0277	インタフェースのプロパティを実装するクラスのプロパティのアクセシビリティが public でない	System.TypeLoadException: Method 'getter' in type ' 型名' does not have an implementation.
CS0418	抽象クラスが sealed または static である	エラーなし
CS0441	static クラスが sealed 指定されている	仕様上発生しない
CS0442	abstract なアクセサーのアクセシビリティが private である	エラーなし
CS0470	メソッドの名前がインタフェースのアクセサーの名前と競合する	エラーなし
CS0500	abstract メソッドが本体を持っている	エラーなし
CS0501	abstract でないメソッドの本体がない	System.InvalidOperationException: Method ' メソッド名' does not have a method body.
CS0502	abstract メソッドが sealed 指定されている	エラーなし
CS0507	メソッドオーバーライドがアクセシビリティを変更している	アクセシビリティが広くなる場合エラーなし，狭くなる場合 System.TypeLoadException: Derived method ' メソッド名' in type ' 型名' cannot reduce access.
CS0508	オーバーライドメソッドの戻り値の型が元のメソッドと一致しない	System.TypeLoadException: Return type in method ' メソッド名' on type ' 型名' is not compatible with base type method ' 元のメソッド名'.

CS0509	基底クラスが sealed 指定されている	System.TypeLoadException: Could not load type ’ 型名’ because the parent type is sealed.
CS0513	abstract メソッドが abstract でないクラスに含まれている	System.InvalidOperationException: Type must be declared abstract if any of its methods are abstract.
CS0515	静的コンストラクターにアクセシビリティが設定されている	エラーなし
CS0525	インスタンスフィールドを持つインタフェースが定義されている	System.TypeLoadException: Instance Field in an Interface.
CS0526	インタフェースのコンストラクターが定義されている	System.InvalidOperationException: Interface cannot have constructors.
CS0527	構造体またはインタフェースがインタフェースでない型を継承している	System.TypeLoadException: Could not load interface ’ 型名’ because it must extend from Object.
CS0529	インタフェースの継承関係が循環参照を含む	System.TypeLoadException: Could not load type ’DynamicInterface’.
CS0534	非抽象クラスが基底クラスの抽象メンバーを実装していない	System.TypeLoadException: Method ’ メソッド名’ in type ’ 型名’ does not have an implementation.
CS0535	インタフェースのメソッドを実装していない	System.TypeLoadException: Method ’ メソッド名’ in type ’ 型名’ does not have an implementation.
CS0545	get アクセサーがオーバーライド出来ない	System.TypeLoadException: Method ’ メソッド名’ in type ’ 型名’ does not have an implementation.
CS0546	seet アクセサーがオーバーライド出来ない	System.TypeLoadException: Method ’ メソッド名’ in type ’ 型名’ does not have an implementation.
CS0547	プロパティまたはインデクサーの型が void 型である	エラーなし
CS0548	アクセサーをもたないプロパティまたはインデクサーが宣言されている	エラーなし
CS0621	virtual または abstract なメンバーのアクセシビリティが private になっている	エラーなし

CS0709	基底クラスが static である	System.TypeLoadException: Could not load type ’ 型名’ because the parent type is sealed.
CS0713	static クラスが基底クラスを持つ	エラーなし
CS0714	static クラスがインタフェースを実装している	エラーなし
CS0736	インタフェースメソッドの実装が static になっているためインターフェースを実装出来ない	System.TypeLoadException: Method ’ メソッド名’ in type ’ 型名’ does not have an implementation.
CS0737	メソッドのアクセシビリティが public ではないためインタフェースメソッドを実装出来ない	System.TypeLoadException: Method ’ メソッド名’ in type ’ 型名’ does not have an implementation.
CS0738	メソッドの戻り値の型が一致しないためインタフェースメソッドを実装出来ない	System.TypeLoadException: Method ’ メソッド名’ in type ’ 型名’ does not have an implementation.
CS0840	abstract でないプロパティが本体を持たない	エラーなし
CS1715	プロパティやインデクサーの型がオーバーライド元と一致しない	System.TypeLoadException: Method ’ メソッド名’ in type ’ 型名’ does not have an implementation.

付録 B

DSL のプログラム

```
1  using System.Reflection;
2  using System.Reflection.Emit;
3  public record Recipe(string Name, bool IsPublic, Type ReturnType, (
       Type type, string name)[] ParameterTypes, params Instruction[]
       Instructions);
4  public record Instruction(string Command, params string[] Args);
5  class EngineFactory
6  {
7      DynamicTypeBuilder typeBuilder { init; get; }
8      Dictionary<string, (Recipe recipe, MethodBuilder methodBuilder)>
           definedRecipe { init; get; } = new();
9      public EngineFactory()
10     {
11         var modBuilder = new RuntimeModuleBuilder(new("Dynamic"));
12         typeBuilder = modBuilder.DefineType("RuntimeEngine",
               TypeAttributes.Class, null, new[] { typeof(IRunner) });
13
14     }
15     public EngineFactory AddRecipe(Recipe recipe)
16     {
17         var attributes = recipe.IsPublic ? MethodAttributes.Public |
               MethodAttributes.Virtual : MethodAttributes.Private |
               MethodAttributes.Virtual;
18         var methodBuilder = recipe.ReturnType == typeof(void) ?
               typeBuilder.AddAction(recipe.Name, attributes, recipe.
               ParameterTypes.Select(((Type type, string name) item) =>
               item.type).ToArray())
19             : typeBuilder.AddMethod(recipe.Name, attributes, recipe.
                   ReturnType, recipe.ParameterTypes.Select(((Type type
                   , string name) item) => item.type).ToArray());
20         definedRecipe[recipe.Name] = (recipe, methodBuilder);
```

```
21          return this;
22      }
23      public RuntimeType Create()
24      {
25          foreach (var (_, recipe) in definedRecipe) Compile(recipe);
26          return typeBuilder.CreateType();
27      }
28      private void Compile((Recipe, MethodBuilder) target)
29      {
30          var (recipe, methodBuilder) = target;
31          var ilGen = methodBuilder.GetILGenerator()!;
32          Dictionary<string, ParameterBuilder> parameters = new();
33          Dictionary<string, LocalBuilder> locals = new();
34          Dictionary<string, (Label startLabel, Label endLabel)>
                labels = new();
35          for (var i = 0; i < recipe.ParameterTypes.Length; i++)
                parameters[recipe.ParameterTypes[i].name] =
                methodBuilder.DefineParameter(i + 1, ParameterAttributes
                .None, recipe.ParameterTypes[i].name);
36
37          Type EmitLoad(string from)
38          {
39              Type? targetType;
40              if (parameters.ContainsKey(from)) { ilGen.Emit(OpCodes.
                    Ldarg, parameters[from].Position); targetType =
                    recipe.ParameterTypes[parameters[from].Position -
                    1].type; }
41              else if (locals.ContainsKey(from)) { ilGen.Emit(OpCodes.
                    Ldloc, locals[from]); targetType = locals[from].
                    LocalType; }
42              else if (int.TryParse(from, out var result)) { ilGen.
                    Emit(OpCodes.Ldc_I4, result); targetType = typeof(
                    int); }
43              else if (from.StartsWith('"') && from.EndsWith('"')) {
                    ilGen.Emit(OpCodes.Ldstr, from[1..^1]); targetType =
                     typeof(string); }
44              else throw new InvalidProgramException($"Recipe '{recipe
                    .Name}' does not have a parameter or local '{from}'
                    ");
45              return targetType;
46          }
47          void EmitLoadInt(string from)
48          {
```

```
49              if (parameters.ContainsKey(from)) ilGen.Emit(OpCodes.
                    Ldarg, parameters[from].Position);
50              else if (locals.ContainsKey(from)) ilGen.Emit(OpCodes.
                    Ldloc, locals[from]);
51              else if (int.TryParse(from, out var result)) ilGen.Emit(
                    OpCodes.Ldc_I4, result);
52              else throw new InvalidProgramException($"Recipe '{recipe
                    .Name}' does not have a parameter or local '{from}'
                    ");
53          }
54          foreach (var instruction in recipe.Instructions)
55          {
56              switch (instruction.Command)
57              {
58                  case "Add":
59                  case "Sub":
60                  case "Mul":
61                  case "Div":
62                  case "Mod":
63                      {
64                          var first = instruction.Args[0];
65                          bool isParameter = false;
66                          if (parameters.ContainsKey(first)) {
                                isParameter = true; ilGen.Emit(OpCodes.
                                Ldarg, parameters[first].Position); }
67                          else if (locals.ContainsKey(first)) ilGen.
                                Emit(OpCodes.Ldloc, locals[first]);
68                          else throw new InvalidProgramException($"
                                Recipe '{recipe.Name}' does not have a
                                parameter or local '{first}' ");
69                          foreach (var arg in instruction.Args.Skip(1)
                                )
70                          {
71                              EmitLoadInt(arg);
72                              ilGen.Emit(instruction.Command switch {
                                    "Add" => OpCodes.Add, "Sub" =>
                                    OpCodes.Sub, "Mul" => OpCodes.Mul, "
                                    Div" => OpCodes.Div, "Mod" =>
                                    OpCodes.Rem, _ => OpCodes.Nop });
73                          }
74                          if (isParameter) ilGen.Emit(OpCodes.Starg,
                                parameters[first].Position);
```

```
75                      else ilGen.Emit(OpCodes.Stloc, locals[first
                            ]);
76                      break;
77                  }
78              case "Show":
79                  {
80                      var first = instruction.Args[0];
81                      Type? targetType = EmitLoad(first);
82                      ilGen.Emit(OpCodes.Call, typeof(Console).
                            GetMethod("WriteLine", new[] {
                            targetType! })!);
83                      break;
84                  }
85              case "Define":
86                  {
87                      var typeName = instruction.Args[0];
88                      var type = Type.GetType(typeName switch { "
                            int" => "System.Int32", "string" => "
                            System.String", var a => a });
89                      if (type is null) throw new
                            InvalidOperationException($"Unknown type
                             '{typeName}'.");
90                      var local = ilGen.DeclareLocal(type);
91                      locals[instruction.Args[1]] = local;
92                      break;
93                  }
94              case "Assign":
95                  {
96                      var assignTarget = instruction.Args[0];
97                      var from = instruction.Args[1];
98                      EmitLoad(from);
99                      if (parameters.ContainsKey(assignTarget))
                            ilGen.Emit(OpCodes.Starg, parameters[
                            assignTarget].Position);
100                     else if (locals.ContainsKey(assignTarget))
                            ilGen.Emit(OpCodes.Stloc, locals[
                            assignTarget]);
101                     else throw new InvalidProgramException($"
                            Recipe '{recipe.Name}' does not have a
                            parameter or local '{assignTarget}' ");
102                     break;
103                 }
104             case "Repeat":
```

```
105                     {
106                         var loopVariable = instruction.Args[0];
107                         var startValue = instruction.Args[1];
108                         var endValue = instruction.Args[2];
109                         var itr = ilGen.DeclareLocal(typeof(int));
110                         locals[loopVariable] = itr;
111                         var startLabel = ilGen.DefineLabel();
112                         var endLabel = ilGen.DefineLabel();
113                         labels[loopVariable] = (startLabel, endLabel
                                );
114                         EmitLoadInt(startValue);
115                         ilGen.Emit(OpCodes.Stloc, itr);
116                         ilGen.MarkLabel(startLabel);
117 
118                         ilGen.Emit(OpCodes.Ldloc, itr);
119                         EmitLoadInt(endValue);
120                         ilGen.Emit(OpCodes.Clt);
121                         ilGen.Emit(OpCodes.Brfalse, endLabel);
122                         break;
123                     }
124                 case "EndRepeat":
125                     {
126                         var loopVariable = instruction.Args[0];
127                         var itr = locals[loopVariable];
128                         ilGen.Emit(OpCodes.Ldloc, itr);
129                         ilGen.Emit(OpCodes.Ldc_I4_1);
130                         ilGen.Emit(OpCodes.Add);
131                         ilGen.Emit(OpCodes.Stloc, itr);
132                         ilGen.Emit(OpCodes.Br, labels[loopVariable].
                                startLabel);
133                         ilGen.MarkLabel(labels[loopVariable].
                                endLabel);
134                         labels.Remove(loopVariable);
135                         break;
136                     }
137                 default:
138                     {
139                         var callTarget = instruction.Command;
140                         ilGen.Emit(OpCodes.Ldarg_0);
141                         foreach (var arg in instruction.Args)
                                EmitLoad(arg);
142                         ilGen.Emit(OpCodes.Call, definedRecipe[
                                callTarget].methodBuilder);
```

```
143                         break;
144                     }
145             }
146         }
147         if (labels.Count != 0) throw new InvalidProgramException($"
                Loop {string.Join(",", labels.Select(l => l.Key))} is
                not closed.");
148         ilGen.Emit(OpCodes.Ret);
149     }
150 }
```