2023 情財第 177 号

2023 年度未踏 IT 人材・育成事業

Python にトランスパイル可能な
静的型付けプログラミング言語の開発

成果報告書

クリエータ：芝山 駿介
担当 PM　：竹迫 良範

2024 年 3 月 8 日

目次

1. 要約

本プロジェクトでは，Python にトランスパイル可能な静的型付けプログラミング言語 Erg の開発を行ない，また開発ツール群 (Language Server，パッケージマネージャ，インストーラ，パッケージレジストリ，パッケージレジストリサイト) も実装した．

2. 背景及び目的

2.1. 背景

Python[*1]は，産業用・教育用途を問わず世界中で幅広く利用されている汎用プログラミング言語である．特に，機械学習やデータ分析の分野での利用が増加しており，その需要は今後も続くと予想される．

しかし，Python は実行時にコードの正当性を検証する動的型付け言語であるため，静的型付け言語に比べてバグの発見が遅れ，開発効率が低下するという傾向がある． また，Python はそのシェアに比べて， 公式の提供する開発環境が不十分であるという問題もある．したがってサードパーティの開発ツールが OSS・商用を問わず多く存在するが，これは開発者にとって選定のコストが大きく，またそれらのツールの互換性による問題が発生することがある．

動的型付け言語のデメリットを解消するべく，動的型付け言語に静的型システムを導入したプログラミング言語を開発するというプロジェクトがある．代表的なものが TypeScript[*2]であり，これは JavaScript[*3]に静的型システムを導入した言語である．TypeScript は IT 系ナレッジサイト stackoverflow が 2023 年に行ったプログラミング言語の人気調査[*4]で 5 位にランクインするなど，大きな成功を収めている． 従って，JavaScript と同じく動的型付け言語である Python に対して静的型システムを導入したプログラミング言語を開発することは，潜

1 https://python.org

2 https://typescriptlang.org

3 https://developer.mozilla.org/ja/docs/Web/JavaScript/

4 https://survey.stackoverflow.co/2023/#most-popular-technologies-language/

在的に大きな需要があると考えられる．

Python をトランスパイル (変換) ターゲットとするプログラミング言語はいくつか存在する[*5][*6]が，その多くは動的型付け言語であり，Python の問題をそのまま引き継いでいる．Python との連携を重視したプログラミング言語として Julia[*7]があり，この言語は静的型付けを部分的に採用しているが，これはあくまで実行効率向上を目的とするところが大きく，動的型付けのコードを混入させることもできてしまう．Python との互換性を謳う静的型付け言語としては，2023 年に発表された Mojo[*8]があるが，この言語もやはり実行効率のために静的型付けを採用していると思われ，Python API の静的型付け化を指向しているとは言い難い．

Python には型アノーテーションの文法があるため，これを用いて擬似的に静的型付けを行おうとするプロジェクト[*9]もある．しかしこのアプローチには大きく 2 つの問題がある．まず 1 つ目は，型アノーテーションはあくまでコメントのようなものであり，実行時には無視されるため，コードの安全性を真の意味で担保することはできないという問題がある．次に，この手のプロジェクトでは，大抵 Python コードの型推論を行わない (行えない) ため，至るところに型アノーテーションを付ける必要があり，大きな負担となる問題がある．Python コードの型推論をベストエフォートで行う検査器[*10]もあるが，Python の文法上自ずと限界がある．

従って，本プロジェクトのように，Python API との互換性を保証しながら，完全な静的型付けの達成を狙うプログラミング言語および型検査器はこれまで確認された限りにおいて存在しない．以上が本プロジェクトを提案するに至った背景である．

---

5 https://hylang.org

6 https://coconut-lang.org

7 https://julialang.org

8 https://mojolang.org

9 https://mypy-lang.org

10 https://github.com/google/pytype

2.2. 目的

前節のような問題を解決するために，本プロジェクトでは Python にトランスパイル可能な静的型付けプログラミング言語 Erg をおよびその開発ツールを開発する．Erg 言語は Python にトランスパイルされるため Python のコード資産をそのまま再利用でき，また静的型システムによる高い静的検証能力を持つ．また開発ツール群をコマンド一つで呼び出せるようにすることで，低い環境構築コストと高い再現性を実現する．

3. プロジェクト概要

本プロジェクトでは，Python にトランスパイル可能な静的型付けプログラミング言語 Erg およびその開発ツール群の開発を行なった．言語仕様は公式サイト[*11]およびリポジトリ[*12]で公開されており，殆どの機能が実装済みである．コンパイラは Rust[*13]で実装されており，MIT[*14]および Apache 2.0 [*15]のデュアルライセンスで公開されている[*16]．OS は Linux，macOS，Windows に対応している． 開発ツール群は，Language Server（4.5 節で説明）， パッケージマネージャ，インストーラ，パッケージレジストリおよびレジストリの登録済みパッケージを確認できる Web サイトを開発した．

4. 開発内容

4.1. コンパイラ

Erg コンパイラは Rust で実装されており，MIT/Apache 2.0 ライセンスで公開されている．本プロジェクト採択前からプロトタイプとして開発を進めてい

11 https://erg-lang.org/the-erg-book/

12 https://github.com/erg-lang/erg/tree/main/doc/JA

13 https://rust-lang.org

14 https://opensource.org/license/mit/

15 https://opensource.org/license/apache-2-0/

16 これは Rust 製のプロジェクトではよく見られるライセンス形態であり， ユーザは MIT ライセンスのように制限の少ないライセンスか，Apache 2.0 ライセンスのように特許権への言及など商用利用者にとって使いやすいライセンスのどちらかを選択できる．

たが，採択後は全体のアーキテクチャの再設計，コンパイラの並列化などに対応したほか，多くのバグ修正と機能追加を行なった．

Rust を開発言語として採用したのは，Rust が言語処理系などを含むシステムソフトウェアの開発言語として設計されたシステムプログラミング言語であり，高い実行効率とメモリ安全性を兼ね備えていると判断したためである．

コンパイラは erg コマンドで呼び出せるが，erg コマンドは後述する Language Server やパッケージマネージャを呼び出すこともできるため，コンパイラ専用のコマンドというわけではない．図 1 にコンパイラのアーキテクチャについての概説図を示す．

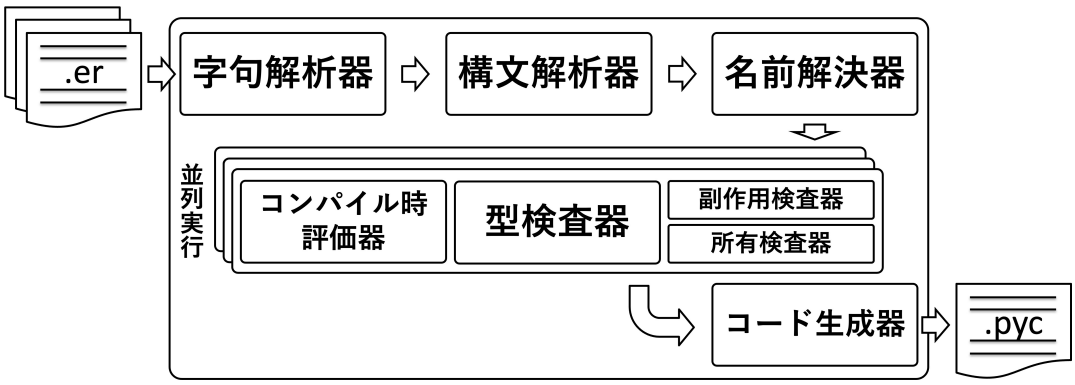

図 1: コンパイラの概説図

これは静的型付け言語のコンパイラとしては一般的な構成であるが，以下に詳しく説明する．なお以下では，字句解析器から型検査器・副作用検査器・所有権検査器までの部分をフロントエンド，コード生成器の部分をバックエンドと呼ぶ．

まず，Erg 言語のソースコードは字句解析器によってトークン列に変換される．トークン列の変換によってスペースやコメントなどの無視されるべき文字列は除去され，インデントの解釈なども行われる．

次に，トークン列は構文解析器によって抽象構文木（AST）に変換される．構文解析器は再帰下降構文解析を行い，文法エラーを検知しても可能な限り解析を続行する．また構文解析のあと，AST 中のパターンマッチなどの構文糖は脱糖される．

名前解決器は，AST からモジュールのインポートを探し出し，そのインポート先のモジュールとの関連付けを行う．そのようにして構築した依存関係グラフに従ってモジュールの意味検査プロセスの実行計画を立てる．

型検査器は，AST を走査して各式の型推論および型検査を行い，AST に型情報が付加された中間表現（HIR）を生成する．コンパイル時評価器は型推論・型検査の過程で必要となる式の評価を行う．型検査器およびその基盤となる静的型システムは非常に重要な部分であるため，4.4 節で詳しく説明する． 本プロジェクト中は実装しなかったが，将来的にはユーザ定義のコンパイル時関数を評価可能として実行時の計算量を削減することも可能になる予定である．

副作用検査器は，HIR を走査して不正な副作用の使用がないか検査する．

所有権検査器は，HIR を走査して参照や move(4.3 節の「所有権」の項を参照) が正当に行われているか検査する．

最後に HIR を結合してコード生成器に渡し， コード生成器はそれを用いて単一の pyc ファイルを生成する． コード生成器はいくつかのバックエンドが実装されているが，現在最も開発が進んでいるのは Python バイトコード (pyc ファイル) を生成するバックエンド (以下，pyc バックエンドと呼称する) である．Python バイトコードは Python インタプリタ（CPython）が実行可能な仮想機械語であり，「Python にトランスパイル可能」というのは主に pyc バックエンドの能力を指している．Python バイトコードは各バージョンによって互換性が保証されないため，pyc バックエンドは現在のところ Python 3.7-3.11 をサポートしている． 直接 Python スクリプトを出力するバックエンドも開発したが，pyc バックエンドの完成度向上を最優先課題としたため， こちらは一部機能が未実装である．また，Rust コードへトランスパイルすることでネイティブコード出力を可能とするバックエンドの開発も行ったが，同様の理由でこちらも基本的機能の実装に留まった．

コンパイルの各ステージで発生したエラーは収集され，一度にユーザに報告される． エラーは該当箇所のコードとともに提示され， エラーメッセージも

ユーザにとって理解しやすいものとなるよう努めた（図 2）.

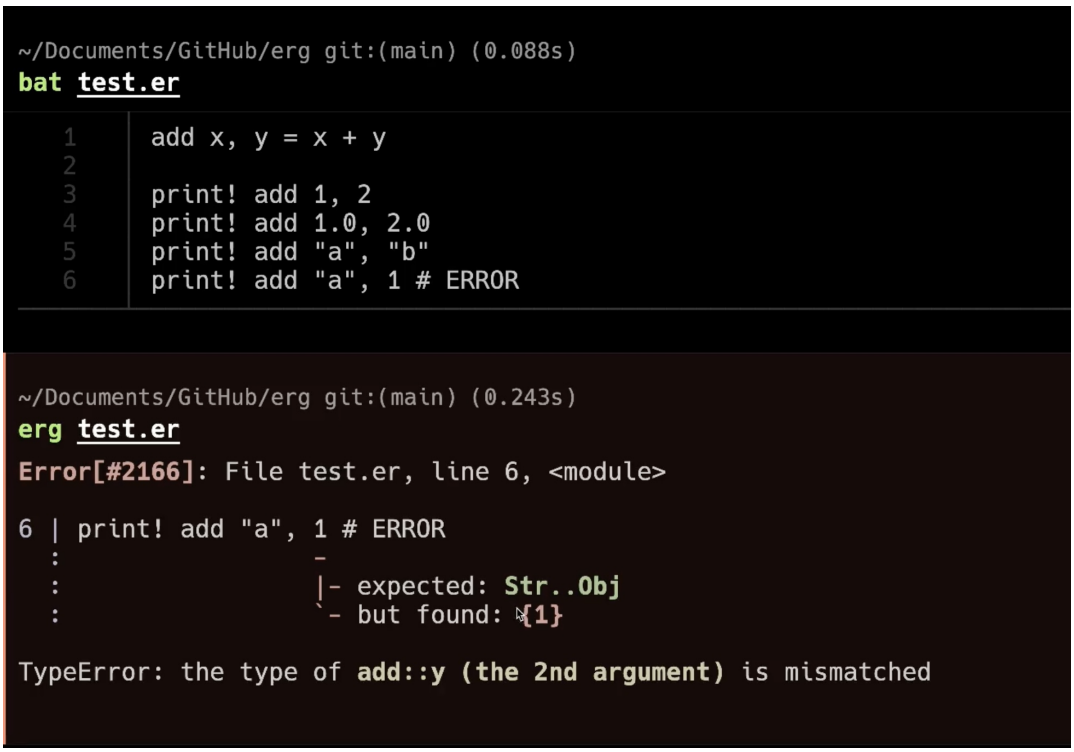

図 2: エラーメッセージの例

なお，erg コマンドで直接ファイル名を指定した場合，生成された pyc ファイルは即座に Python インタプリタに渡され実行される (実行後ファイルは残らない)．pyc ファイルのみを出力したい場合は，erg compile サブコマンド，検査だけを行いたい場合は erg check サブコマンドを使う．

4.2. Python パッケージとしての Erg コンパイラ

Erg コンパイラは，Python パッケージとして Python インタプリタが動的に呼び出すこともできるようにした． これにより，Erg モジュールを実行時に読み込んだり，Erg コードの AST を直接操作したりすることが可能になる．Python でのコード例を以下に示す．

```python
# Python
import erg_compiler
from erg_compiler import erg_parser
```

```
compiler = erg_compiler.Compiler(deps=[])
code = compiler.compile(".i = 0", "exec")
exec(code)
assert i == 0

mod = erg_parser.parse(".j = 1")
ast = erg_parser.ast.AST.new("test", mod)
test = erg_compiler.exec_ast(ast)
j = test.__dict__["j"]
assert j == 1
```

API の型定義も行ったので，Erg から Erg コンパイラを操作することも可能である．Python パッケージ化には pyo3[*17]を用いた．これは Rust と Python の間での相互運用を可能にするライブラリである．

4.3. 言語仕様

Erg は以下の設計思想に基づいて設計されている．

- 強い静的型付け
- ミニマルな基礎文法
- 高い表現能力
- Python API との互換性
- 高い開発効率

以下では Erg の言語仕様を概説する．詳細な資料はインターネット上で公開されている[*18]のでそちらも参照されたい．

変数定義・関数定義

変数は Python と同様の文法で定義できるが，再代入はできない．これは

17 https://pyo3.rs

18 https://erg-lang.org/the-erg-book/

関数型言語ではよく見られる仕様で，再代入がないことでプログラムの挙動を追いやすくなるといったメリットがある．

```
i = 1
i = 2 # NameError: variable i cannot be as↵
 ↳signed more than once
```

大文字で始まる変数は定数を表す．定数はコンパイル時に値が確定していなければならない．

```
TWO = 1 + 1
PI = 3.14
```

変数は明示的に型指定できるが，型推論が行われるため省略することもできる．

```
i: Int = 1
a: Array(Int) = [1, 2, 3]
```

関数は Python とは少し異なる文法で定義される． これは Haskell などの関数型言語の影響を受けている． 関数も変数と同様に型指定できるほか，型推論によって省略することもできる． 引数リストの () は戻り値型を指定しない場合省略できる．

```
add x, y = x + y
iadd(x: Int, y: Int): Int = x + y
```

デフォルト引数は以下のような文法でサポートしている．

```
isum iterable: Iterable(Int), start := 0 =
    sum(iterable, start)
```

多相関数の明示的な型指定は以下のような文法でサポートしている．

```
id|T|(x: T): T = x
add|A <: Add(R)|(x: A, y: R): A.Output =
    x + y
```

関数呼び出し

関数呼び出しは Python とほぼ同様の文法で行うことができる．呼び出しの () は意味が一意に定まる場合省略できる．

```
f(1, 2)
f 1, 2
```

キーワード引数も以下のような文法でサポートしている．

```
sum([1, 2, 3], start:=1)
```

Erg のサブルーチンの中には，副作用を許容するものと許容しないものがある．副作用を許容するものは手続きまたはプロシージャ（procedure）と呼ばれ，それ以外のものは関数（function）と呼ばれる．Python の関数は手続きに対応するので注意されたい．プロシージャはポストフィックスとして！をつける必要がある．これにより，コードに副作用があるのかないのかを一目で判断することができる．

```
add 1, 2 # no side-effect
print! 1, 2 # side-effect
```

組み込みサブルーチン

Erg には，Python の組み込み関数に加えて Erg 独自の組み込みサブルーチンが実装されている．以下に主要なものを示す．

- print!: 標準出力に出力する

- open!: ファイルを開く
- len: コンテナオブジェクトの長さを返す
- sum: コンテナオブジェクトの要素を合計する
- range: range オブジェクトを生成する
- zip: 複数のコンテナオブジェクトをまとめる
- map: 関数をコンテナオブジェクトの各要素に適用する
- exit: プログラムを終了する
- for!: コンテナオブジェクトの各要素に対して処理を行う
- if!: 条件によって処理を分岐する
- while!: 条件が真の間処理を繰り返す
- match!: パターンマッチングを行う

特筆するべきは，for! や if!，while!，match! などの制御構造が Erg では単なる手続きである点である．以下がその使用例である．

```
for! [1, 2, 3], i =>
    print! i

if! True:
    do!:
        print! "True"
    do!:
        print! "False"

while! True, do!:
    print! "loop"

match! 1:
    1 => print! "one"
```

```
15      2 => print! "two"
16      _ => print! "other"
```

これにより，Erg ではこれらの手続きを組み合わせて容易に新しい制御構造を定義することができる．

組み込み型

組み込み型は、Python の組み込み型とほぼ同じものがサポートされている．主要なものを以下に示す．

- 整数型 (Int)
- 浮動小数点数型 (Float)
- 真偽値型 (Bool)
- 文字列型 (Str)
- None 型 (NoneType)
- バイト列型 (Bytes)
- 配列型 (Array)
- 辞書型 (Dict)
- タプル型 (Tuple)
- 集合型 (Set)
- 関数型 (Func)

このほか，Erg 独自の組み込み型として以下のものがサポートされている．

- 自然数型（Nat: 0 以上の整数で，Int のサブタイプ）
- プロシージャ型（Proc: 手続き，すなわち副作用を許容するサブルー

チンの型，Python の関数に対応するのはこちら）

- レコード型（Record）

レコードは JavaScript のオブジェクト記法に似たものであり，以下のような文法で定義できる．

```
p = { .x = 1; .y = 1 }
assert p.x == 1 and p.y == 1
```

また，Int や Array などにはそのサブタイプとして内部可変性を持つ型が存在する（Int!，Array! など）．Erg では型にポストフィックスとして！を付けることで内部可変性を指定できる．つまり！の付かない型とサブルーチン (関数) のみを使えば，純粋関数型言語として Erg を使うことが可能である．

クラスの定義

クラスの定義は，以下のような文法で行うことができる．

```
Point2D = Class {
    x = Int
    y = Int
}
```

メソッドの定義は，以下のような文法で行うことができる．

```
Point2D.
    norm(self) = self.x ** 2 + self.y ** 2
```

このようにして定義されたクラスは以下のようにしてインスタンス化できる．

```
p = Point2D.new {x = 1; y = 2}
```

```
2 assert p.norm() == 5
```

new はインスタンスを生成するスタティックメソッド（コンストラクタ）であり，ユーザがカスタムすることもできる．

```
1 Point2D.
2     new(x: Int, y: Int) =
3         Point2D {x = x; y = y}
4
5 p = Point2D.new(1, 2)
```

所有権

Erg にはオブジェクトの所有権の概念が存在する． これは Rust に影響を受けたものであるが，若干簡略化されている．Erg では，可変オブジェクトの所有権はただ一つである．所有権が移動した後の変数にアクセスするとコンパイルエラーが発生する．以下がそのコード例である．空の可変配列は ![] で生成するが，これを束縛する変数は，ただ一つのみに制限される．

```
1 arr = ![]
2 arr2 = arr
3 print! arr2 # []
4 print! arr # MoveError: arr was moved in li↵
  ↳ne 2
```

Python の奇妙な挙動としてよく知られている， リストの代入が参照のコピーであるという挙動は， この所有権の概念のために Erg では起こらない．

```
1 # Python
2 list = []
3 list2 = list
```

```
4 list2.append(1)
5 print(list) # [1]
```

4.4. 静的型システム

本節では Erg の静的型システムについて説明する．前提として，Python API は動的型付けの前提で設計されている．これをなるべく使用感を損なわず静的に型付けするためには，従来の静的型付け言語のものよりも表現力の高い，強力な静的型システムが必要である．このため Erg 言語では，以下のような特筆すべき型システムを持つ．

- トレイト
- 制御構造に基づく型解析（control flow based type analysis）
- オーバーロード型
- 多相関数型
- 構造的部分型
    - Structural 型
- 直和型
- 交差型
- 依存型
- 篩型

トレイトは複数の異なる型に共通する性質を表現し， 統一的に扱うための型である．Python の抽象基底クラスに似ているが， 後述する直和型や交差型などとして合成ができるなど， より柔軟である．Erg ではトレイトを用いてメソッドのインターフェース化や演算子のオーバーロードなどを行うことができる． 以下ではオブジェクトの hash 値を計算する __hash__ メソッドを統一的に扱うトレイトとして Hash を定義する例を示す．

```
# Hashは標準で定義されているトレイト
Hash = Trait {
    # Selfはトレイトを実装する具体的な型を表す
    __hash__: (self: Self) -> Int
}

myhash h: Hash = h.__hash__()
assert myhash(1) == 1
print! myhash("a") # -6206635357184620365
```

対象のクラスに「実装」することで，そのクラスはトレイトに含まれる型として扱えるようになる．後述する Structural 型を用いれば実装なしでもメソッドの存在だけでトレイトに含まれる型のように扱える．

```
C = Class { x = Int }
C|<: Hash|.
    __hash__ self = self.x.__hash__()
```

オーバーロード（多重定義）型は，引数の型によって戻り値の型が変わる関数をアドホックに表現するための型である．Python API には多くの（暗黙的な）オーバーロード関数が存在するためにこのオーバーロード型が存在しているが，あくまで Python API を型付けするためのものであり，Erg 側で定義する関数ではオーバーロードはできない．代わりに多相関数を用いる．以下では，オーバーロードされた関数の使用例を示す．

```
f: (Int -> Float) and (Str -> Str)
f = f(1) # f: Float
s = f("a") # s: Str
```

多相関数型は，型変数と呼ばれる型上を動く変数を用いて抽象的な引数の型，戻り値の型を表現できる関数の型である．トレイトを用いて型変数に制約を与えることもできる．以下は，恒等関数 id と 2 引数を加算する関数 add の定義と使用例である．

```
id: |T|(T) -> T
id x = x
i = id(1) # i: Int
s = id("a") # s: Str

add: |A <: Add(R), R|(x: A, y: R) -> A.Output
add x, y = x + y
n = add(1, 2) # n: Nat
s = add("a", "b") # s: Str
```

構造的部分型は，型の構造が一致していれば異なる型であっても同一視する型システムである．いわゆるダックタイピングを静的に型付けすることができる．構造的部分型の対義語として公称的部分型があるが，こちらはクラスの継承関係やトレイトの実装などによって明示的に部分型関係を宣言する．構造的部分型の方が強力であるが，公称的部分型の方が型検査の効率性が高い．Erg の部分型付けは基本的に公称的部分型だが，Structural 型を用いてオプショナルに実現可能である．例えば，Str 型の属性 name を持つオブジェクトは，すべて Structural { .name = Str } に含まれる．以下がその使用例である．

```
name n: Structural { .name = Str } = n.name

D = Class { .name = Int; .id = Nat }
D.
    new name, id = D { .name; .id }
E = Class { .name = Str; .id = Nat }
E.
    new name, id = E { .name; .id }

d = D.new 1, 2
e = E.new "a", 3

print! name(1) # ERROR
print! name(d) # ERROR
```

```
15 print! name(e) # "a"
```

直和型は，複数の型のうちどれかの要素であることを要求する型である．T 型と U 型の直和型，すなわち T 型か U 型の可能性がある項の型は T or U 型と表現される．これにより，例えば，異なる型の値を一つの変数に束縛することができる．

```
1 stringify x: Int or Str = match x:
2     i: Int -> str i
3     s: Str -> s
4
5 print! stringify 1 # "1"
6 print! stringify "a" # "a"
```

交差型は，複数の型すべての要素であることを要求する型である．T 型と U 型の交差型，すなわち T 型と U 型の両方の要素である項の型は T and U 型と表現される．これにより，例えば一つの型変数に対して複数のトレイトを制約として与えることができる．例えば Erg の辞書のキーの型は Eq トレイトと Hash トレイトの両方を実装している必要があるが，これは交差型を用いて以下のように表現される．

```
1 dict1|K <: Eq and Hash| k: K, v = {k: v}
2
3 dict1 1, 2 # {1: 2}
4 dict1 1.0, 2 # ERROR
```

制御構造に基づく型解析は，型の絞り込みを実現する機能である．例えば，i は None の可能性がある整数型（Int or NoneType）であるとする．この場合，i + 1 はエラーになる．

```
1 i: Int or NoneType
2 i + 1 # TypeError: the type of `+`::lhs (the 1 ↵
```

```
 ↳ st argument) is mismatched
# expected: Add({1}..Obj)
# but found: Int or NoneType
# hint: cannot add Int and NoneType
```

しかし i が None でないことを確かめられれば，i + 1 は正当な式である．そこで，if や assert などの制御構造を用いて i が NoneType でないことを確かめることで，そのスコープ内では i 型を Int に絞り込むことができる．これが制御構造に基づく型解析（control flow based type analysis）である．特に assert を使ってその場で型を絞り込む機能を assert casting と呼ぶ．

```
res = if i != None, do:
    i + 1 # OK
# iがNoneならばres == Noneなので,res: Int or None
 ↳ Type
res: Int or NoneType
print! res # 2
assert res != None
res: Int
```

依存型は多相型の中でも値をパラメータに持つ型である．通常の多相型では以下のように内部に型を持つことができるが，値は持つことができない．

```
a: Array(Int)
```

しかし依存型では，以下のように値を持つことができる．この場合は，依存型を用いて配列の長さを表す値を持つ配列型を表現している．

```
a: Array(Int, 3)

concat|T: Type, M: Nat, N: Nat|(l: Array(T, M
```

```
    ↳ ), r: Array(T, N)): Array(T, M + N) = l + r
4
5   l: Array(Nat, 6) = concat [1, 2, 3], [4, 5, 6]
6   assert l == [1, 2, 3, 4, 5, 6]
```

依存型は，多相関数型や篩型と組み合わせて使うこともできる．以下は，数値計算ライブラリ numpy の多次元配列型 ndarray の reshape メソッドを Erg の型システムで型付けした例である．reshape の前後で要素数が変わらないことが静的に保証されている．

```
1   .NDArray(T, S).
2       reshape: |T, Old: [Nat; _], S: {A: [Nat; ↵
        ↳ _] | A.prod() == Old.prod()}|(
3           self: .NDArray(T, Old),
4           shape: {S},
5       ) -> .NDArray(T, S)
```

4.5. Language Server

Language Server（Erg Language Server）も Rust で実装されており，コンパイラと同じライセンスで公開されている．Language Server とは，対象言語のコーディングを支援する機能 (エラー表示，補完機能，定義ジャンプなど) をエディタに提供するためのソフトウェアである．Language Server Protocol（LSP）[*19]と呼ばれるプロトコルに対応することでエディタとの通信を可能にする．

Language Server も本プロジェクト採択前からプロトタイプとして開発を進めていたが，採択後は並列化，バグ修正と機能追加を行なった．Erg Language Server は Erg コンパイラのフロントエンドと連携することで動作する．erg server サブコマンドで起動することができるが，基本的にこのコマンドはエディタが Language Server との通信を確立するために使うものであり，ユーザが直接利用するものではない．

19 https://microsoft.github.io/language-server-protocol/

4.6. パッケージマネージャ

パッケージマネージャ（開発コードネーム：poise）は Erg 自身を用いて実装し，コンパイラと同じライセンスで公開されている．パッケージマネージャは erg pack サブコマンドで起動することができる．

機能としては

- パッケージの作成・初期化（erg pack init）
- パッケージのインストール（erg pack install）
- パッケージのアンインストール（erg pack uninstall）
- パッケージのアップデート・依存関係の解決（erg pack update）
- パッケージのビルド（erg pack build）
- アプリケーションパッケージの実行（erg pack run）
- パッケージのテスト（erg pack test）
- パッケージのレジストリへの登録（erg pack publish）

が可能である．

パッケージの作成・初期化機能では，パッケージの雛形ディレクトリやパッケージ情報を記述するファイル（package.er）を生成する．

パッケージのインストール機能では，指定されたパッケージをレジストリから取得し， 依存関係を解決してコンパイルし， 出力された疑似実行可能ファイルを bin ディレクトリに配置する．パッケージの指定がない場合はカレントディレクトリのローカルパッケージをインストールする．

パッケージのアンインストール機能では，指定されたパッケージをローカルから削除する．

パッケージのアップデート・依存関係の解決機能では，レジストリの最新情報を読み込んで，カレントディレクトリのローカルパッケージの依存関係を最

新バージョンに更新する．

パッケージのビルド機能では，カレントディレクトリのローカルパッケージをビルドする．

アプリケーションパッケージの実行機能では，カレントディレクトリがアプリケーションパッケージならばビルドして実行する．

パッケージのテスト機能では，カレントディレクトリのローカルパッケージのテストを実行する． テストは tests 以下に配置されたテストスクリプトを実行することで行われる．

パッケージのレジストリへの登録機能では，カレントディレクトリのローカルパッケージを検証し，レジストリに登録する（登録には GitHub アカウントと GitHub CLI のインストールが必要となる）． パッケージマネージャ自身もアプリケーションパッケージとしてレジストリに登録されており，poise を用いて poise 自身をビルド・アップデートすることができる．パッケージ情報については package.er という Erg ファイルに記述する． これはパッケージの名前，バージョン，依存関係などを記述する一種の DSL(ドメイン特化言語) となっている．パッケージのレジストリへの登録については、後述のパッケージレジストリの項で詳述する．

4.7. インストーラ

インストーラは，Erg 言語のコンパイラ， パッケージマネージャ等を一括でインストールするためのスクリプトである． インストール先コンピュータのアーキテクチャに適合するコンパイラのバイナリと標準ライブラリを GitHub release から取得する． バイナリは最新バージョンのリリースによって CI がトリガーされ GitHub release に自動でアップロードされるようにした．安定版のほか，開発版もインストール可能なオプションを実装した．また，パッケージマネージャは，ソースコードを取得しコンパイルすることでインストールする．インストーラは Erg で実装されているが，Erg コンパイラがインストールされていないコンピュータ上では実行できないため，Python スクリプトバックエンドを使用して Python スクリプトに変換し配布している．

4.8. パッケージレジストリ

パッケージレジストリはコードホスティングサイト GitHub を用いて構築した．パッケージマネージャは，パッケージレジストリのリポジトリへ圧縮されたパッケージを Pull Request として送信し，メンテナの確認ののち merge することでパッケージの登録を行う．パッケージの merge が完了すると，CI がトリガーされパッケージレジストリサイトの更新が自動的に行われる．

4.9. パッケージレジストリサイト

パッケージレジストリに登録されたパッケージの情報を閲覧するためのウェブサイトを構築した．この Web サイトは https://package.erg-lang.org で公開されている．図 3 に Web サイトのスクリーンショットを示す．

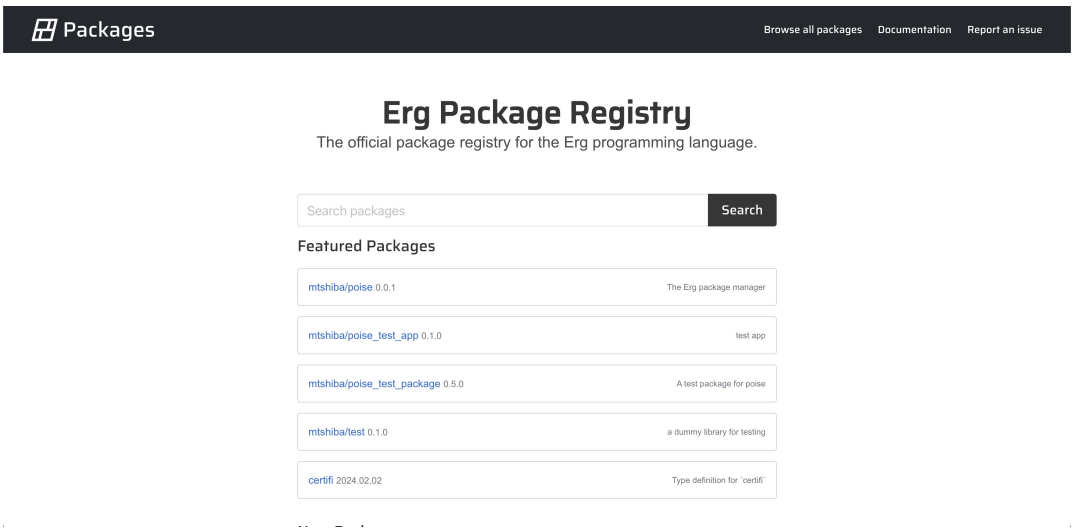

図 3: パッケージレジストリサイトのスクリーンショット

パッケージレジストリサイトでは，パッケージの検索やメタデータの閲覧が可能である．Web サイトは GitHub Pages を用いて構築されており，パッケージレジストリのリポジトリに変更があると CI がトリガーされ，Web サイトの html コードが生成される．検索などの処理はクライアントサイドで JavaScript を用いて行われるようにしており，現在のところ検索用サーバーなどは用意していない．

4.10. 著名な Python パッケージの型定義パッケージ群

Erg コンパイラは Python の組み込み API や標準ライブラリの型定義を同梱しているが，サードパーティの Python パッケージの型定義は Erg パッケージとしてユーザにオンデマンドで提供されるか，ユーザ自身で型定義ファイルを記述する方式になっている．

このうち著名なサードパーティの Python パッケージについては， 型定義パッケージを用意しレジストリに登録した．例としては，

- numpy
- pandas
- matplotlib
- requests
- torch
- torchvision
- urllib3
- erg_compiler (Python パッケージとしての Erg コンパイラ)

などである． これらの型定義は pytypes というリポジトリ[*20]で管理されている．

5. 開発成果の特徴

本プロジェクトでは Python にトランスパイル可能な静的型付け言語 Erg の開発を行った．Erg は Python にトランスパイル可能・あるいは互換性を持つとされる他の言語と比べ， 完全に静的型付けであり，Python API の型定義を自前で行っている点が特徴である． また，Erg の強力な型システムにより，Python API はその使用感をほとんど損なわない形で型定義されている． 先行する Hy や Coconut，Julia などの言語は動的型付けであり，本プロジェクトの

20 https://github.com/erg-lang/pytypes

解決したい問題に応えるものではない．Mojo は Python との互換性を謳う静的型付け言語であるが，実際の API は Mojo 独自のものが多く，また開発環境も整備されていない．

本プロジェクトではコンパイラ本体に加えて，Language Server，パッケージマネージャ，インストーラ，パッケージレジストリ，パッケージレジストリサイトなどの周辺ツールも開発した．Erg 言語は単なる概念実証ではなく，実用プログラミング言語としての基盤を持っていると言える．

6. 今後の課題

6.1. マクロ機構の実装

今後 Erg 言語を Python ユーザ等に広く認知・利用してもらうにあたって，Erg の基礎文法が Python のそれとは表面的に乖離していることがユーザビリティの面で問題になりうる．例えば以下のような問題がある．これは Python の for 文の例である．

```
for i in range(10):
    print(i)
```

同様のコードを Erg で書くと以下のようになる．

```
for! range(10), i =>
    print!(i)
```

基本的構造は同じだが，イテレーション可能オブジェクトとイテレーション変数の位置が逆転している．for 文を導入せず関数呼び出しだけでループを実現しているという点では Erg の文法を好むユーザもいるかもしれないが，Python に慣れたユーザにとっては入力しづらいコードになっている．しかし Erg はミニマルな基礎文法で豊富な機能を提供することを指向しており，for 文のような制御構造を追加することは基本方針に反する．

そこで，なるべく Python の文法に近い形で Erg コードを書けるようにするため，マクロ機構を導入することを検討している．現在構想されているマクロ

機構は Scala[*21]や SATySFi[*22]のような言語の衛生的マクロ機構を参考にしており，マクロの定義・呼び出しにも型検査を行うことで生成されるコードが必ず文法的・型的に正しいことを保証する．

6.2. ネイティブコードバックエンドの実装

4.1 節で述べたように，Erg コンパイラは Python バイトコードを生成するバックエンドが現在の主要なバックエンドであるが，今後コンパイルターゲットを増やしていく予定である．特に本プロジェクト期間中は基本的機能の実装に留まったネイティブコードバックエンドの開発を進め，公開する予定である．

6.3. コンパイラの最適化

Erg コンパイラのパフォーマンスは，パッケージマネージャのソースコード（1000 行弱）を数百ミリ秒でコンパイルできる程度である．実用的には問題ないと言えるが，まだまだ最適化の余地がある．簡易的なプロファイリングでは名前解決や型変数の除去処理がホットスポットであることが判明しており，名前解決器や型検査器の最適化を行うことでさらなるコンパイル速度の向上が期待できる．

6.4. 型定義パッケージの拡充

前章で述べたように，Python の著名なサードパーティパッケージの型定義パッケージをレジストリに登録したが，これらは膨大な量の API があるので，本プロジェクト期間中は主要な API に絞って型付けを行った．今後はより多くの API に対応する予定である．また，著名な Python パッケージの中でまだ型定義パッケージを用意できていないものについても，今後対応していく予定である．

6.5. 広報活動の実施

Erg 言語はまだまだ発展途上であるものの，本プロジェクトの完遂によって実用プログラミング言語としての基盤が整ったと言えるので，ユーザ獲得へ向

21 https://scala-lang.org

22 https://github.com/gfngfn/SATySFi

けて広報活動を実施していきたいと考える．Python や Rust などプログラミング言語系のコミュニティの主催するカンファレンスに参加して登壇したり，ある程度 Erg を学習したいというユーザが集まればワークショップを開催などしたいと考えている．

7. 実施計画書内容との相違点

当初本プロジェクトでは，採択以前から既に pyc バックエンドを含めたコンパイラのプロトタイプが存在していたために，プロジェクト期間中はネイティブコードバックエンドと Python インタプリタとのバインディング機構の実装に大きな重点を置く予定であった．しかしパッケージマネージャの実装を進めるうちに pyc バックエンドや Language Server に多くのバグや改善可能点が見つかり，これらの修正・改善を施して実用プログラミング言語としてのクオリティを向上させることに重点を移した．

8. 成長の自己分析

実装能力の面での成長に関しては，コンパイラや Language Server の再設計によって，コンパイラ実装は勿論のこと，並列処理など低レイヤーの知識を全般的に深めることができた．

また Erg 言語の開発自体は本プロジェクト採択前から始めていたが，個人プロジェクトであったために欠けていた経験を得ることができた．例えば，定期報告の日時から逆算して実装機能の優先度を定め，それに基づいて開発を進める能力が向上した．また，各種会議や成果報告会での発表を通じて，他者に技術を噛み砕いて伝えるプレゼン能力が向上した．またそのようなプレゼンの中で，型理論やプログラミング言語理論について今までは実装レベルの知識しかなかったが，他者に淀みなく説明するためには理論的な知識も必要であることに気づき，それらの知識を更に深めたいと思うようになった．

9. 付録

9.1. 用語説明

コンパイラ

あるプログラミング言語で書かれたプログラムを，コンピュータが解釈・実行できる低水準言語あるいはそれに近い中間言語に変換するプログラム．主な処理系がコンパイラである言語をコンパイラ言語と呼ぶ．これに対して，主な処理系がインタプリタである言語をインタプリタ言語と呼ぶが，近年はインタプリタ言語の中でも実行時コンパイラ（JIT コンパイラ）を持つものがあり，厳密な区別があるわけではない．実際 Erg も Erg コンパイラによって Python バイトコードに変換されるが，Python バイトコードは Python インタプリタによって実行されるため，両方の性質を持つ．

トランスパイラ

あるプログラミング言語で書かれたプログラムを，別のプログラミング言語で書かれたプログラムに変換するプログラム．広い意味でのコンパイラの一種である．

API

Application Programming Interface の略語．プログラムの機能を利用するためのインターフェースのことである．プログラミング言語の文脈で API という場合，通常はライブラリやフレームワークが提供する関数やクラスなどを指す．

Python

Guido van Rossum によって開発された汎用プログラミング言語．現在は Python Software Foundation の主導によって開発が続けられている． 特にデータ分析や機械学習の分野で広く使われている．

CPython

Python の標準実装であり，Python Software Foundation によって開発が続けられている．CPython 以外の Python 実装としては，PyPy や Jython などがある．Erg の pyc バックエンドは現在のところ CPython のみサポートしている．

Rust

Graydon Hoare によって開発されたシステムプログラミング言語．現在は Rust Foundation の主導によって開発が続けられている．メモリ安全性と実行効率の両立を重視して設計されている．

構文糖

構文糖または糖衣構文（syntactic sugar）とは，言語機能としては必須でないものの，プログラムの記述を簡潔にするためにある構文のことである．構文糖はより基本的な構文に変換・分解される．例えば Erg では，パターンマッチは変数代入と条件分岐の組み合わせに分解される．

AST

抽象構文木（Abstract Syntax Tree）の略語．プログラムの構文を木構造で表現したものである．

HIR

高レベル中間表現（High-level Intermediate Representation）の略語．多くのコンパイラは，AST とコンパイルターゲットの間には大きな構造的差異があるために，直接 AST からコンパイルターゲットを生成することはせず，間にいくつかの中間表現を経由する．Erg は AST と Python バイトコードの間に HIR を経由する．HIR は，大まかに言えば型情報付きの AST である．その他，いくつかの構文糖が脱糖されている．

LSP

Language Server Protocol の略語．Language Server とエディタの間で通信を行うためのプロトコルである．Language Server は，対象言語のコーディングを支援する機能（エラー表示，補完機能，定義ジャンプなど）をエディタに提供する．

9.2. 関連 Web サイト

- 公式サイト：https://erg-lang.org
- パッケージレジストリサイト：https://package.erg-lang.org

- GitHub リポジトリ：https://github.com/erg-lang/erg
- 公式ドキュメント：https://erg-lang.org/the-erg-book/