2023年度未踏 芝山PJ開発進捗報告(10月)

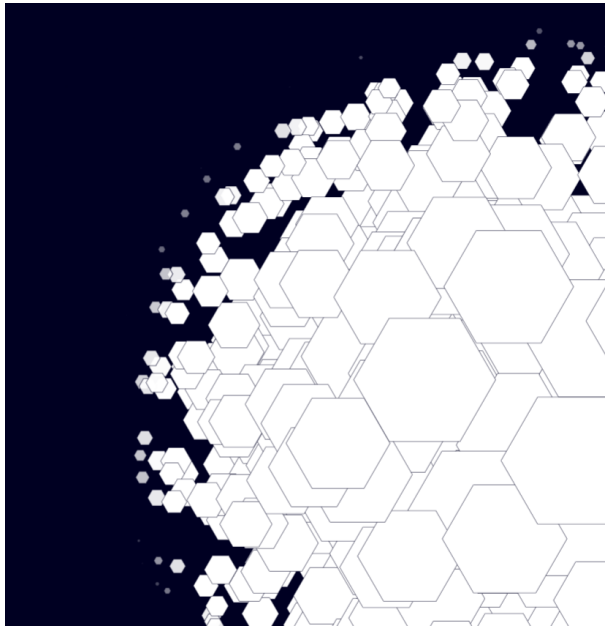
Pythonにトランスパイル可能な静的型付けプログラミング言語の開発

芝山駿介

自己紹介

芝山駿介

早稲田大学先進理工学部物理学科4年
大学では場の量子論、量子情報などを
勉強しています



背景

Pythonは人口に膾炙しているが、とても奇妙な言語

```
Python 3.11.0 (main, Jul 12 2023, 00:27:36) [Clang
Type "help", "copyright", "credits" or "license" f
>>> class C:
...     x: int
...     def __init__(self, x):
...         self.x = x
...
>>> c = C(1)
>>> c == c # must be True
True
>>> C(1) == C(1) # !?
False
>>>
```

なぜこのような挙動になるか？

A. `__eq__` を実装していないから

`__eq__` が実装されていない場合、オブジェクトのIDで比較を行う

下のようにする必要があった

```
class C:
    x: int
    def __init__(self, x):
        self.x = x
    def __eq__(self, other):
        return self.x == other.x
```

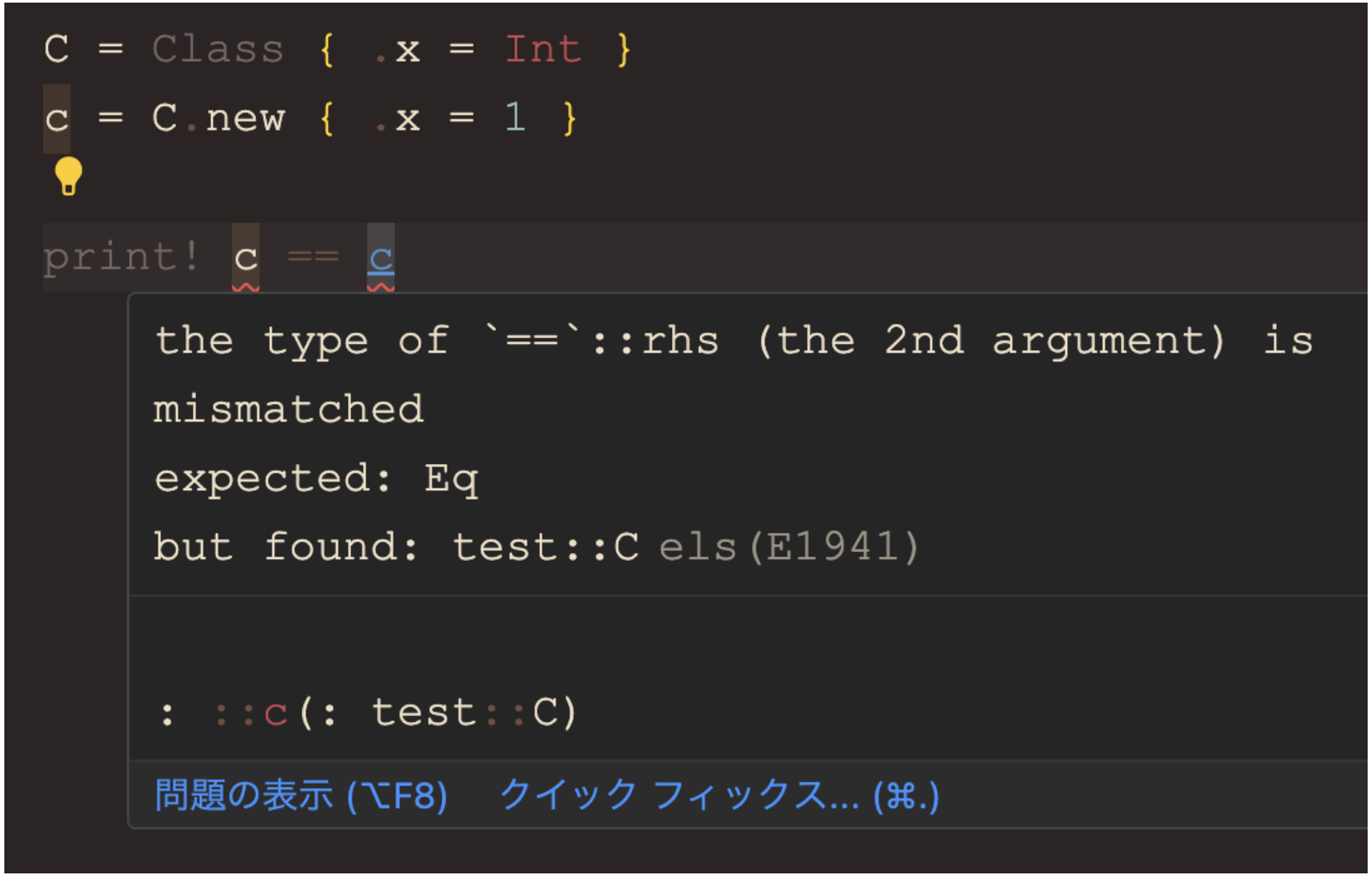
でも、`__eq__`を実装していないならエラーにしてほしい
(願わくば、実行前に)

というか、Pythonに静的型システムが欲しい

言うなればPython版TypeScript

理想

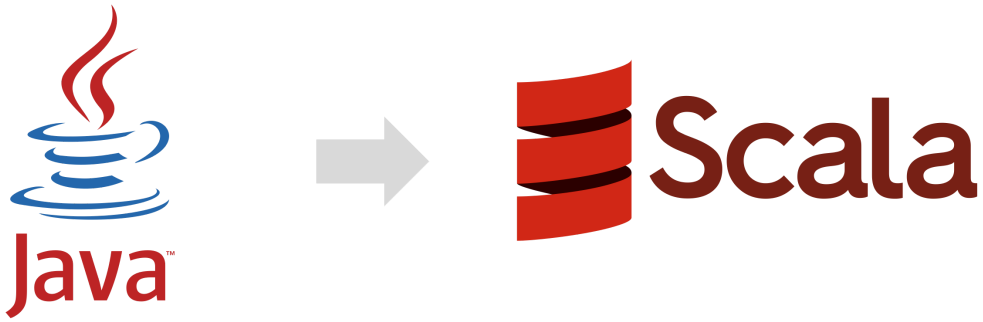
先ほどのような場合は、変数の型が比較処理を実装しているかチェックできれば良い

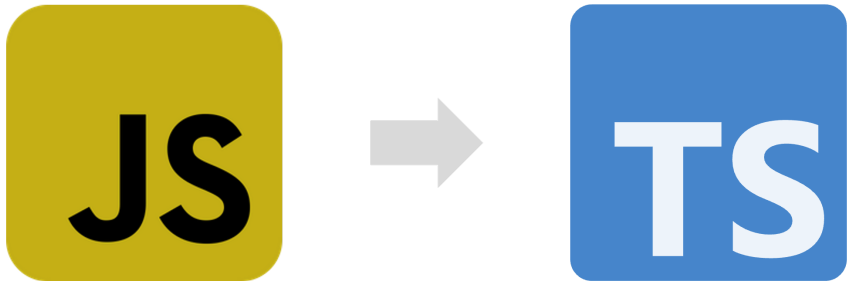


もちろん、`==`の静的検査が出来る言語ぐらい山ほどある

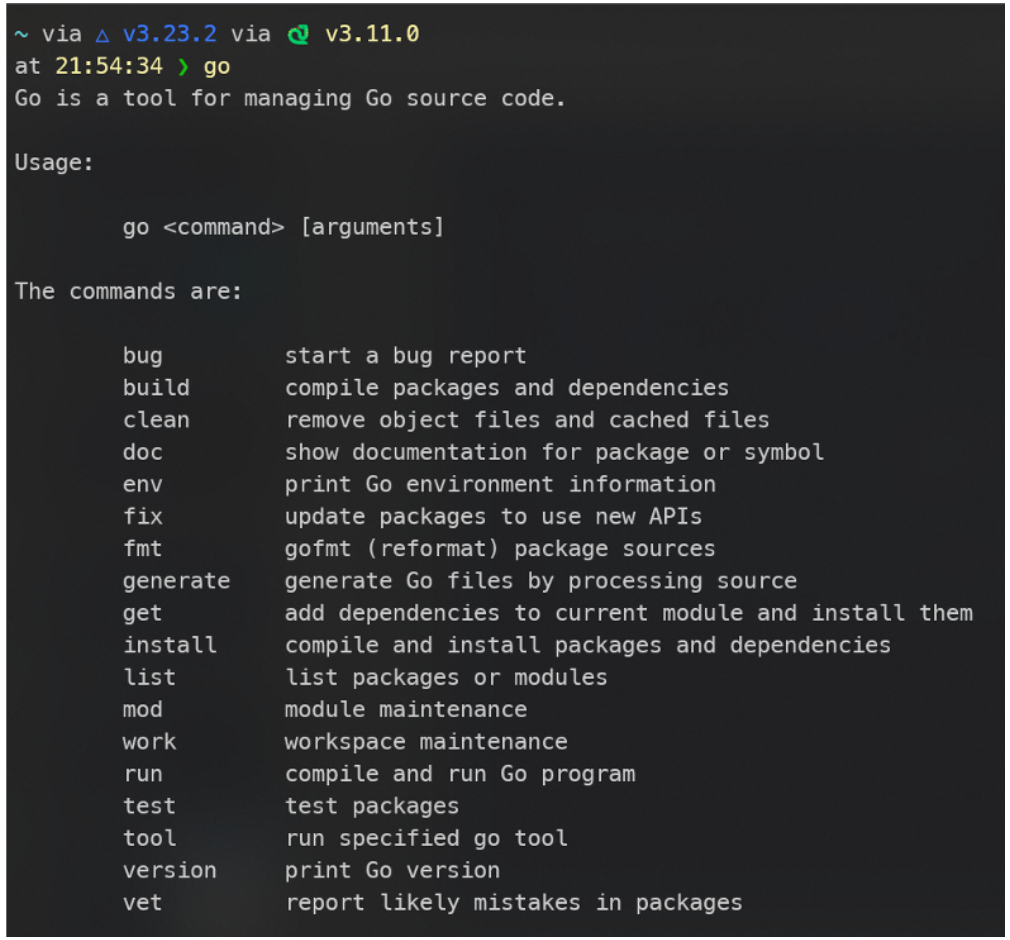
ただ新しい言語を作るだけではダメ
一般にソフトウェアの使用言語の移行は大きなコストが伴う

既存言語=Pythonのコード資産がそのまま使えるような言語が欲しい





また、Python代替を目指すならエコシステムも揃っていて欲しい(パッケージマネージャ, フォーマッタ, Language Server, etc.)
→ Pythonの公式エコシステムはあんまり出来も良くないので、後発の強みを活かして改善されているとなお良い

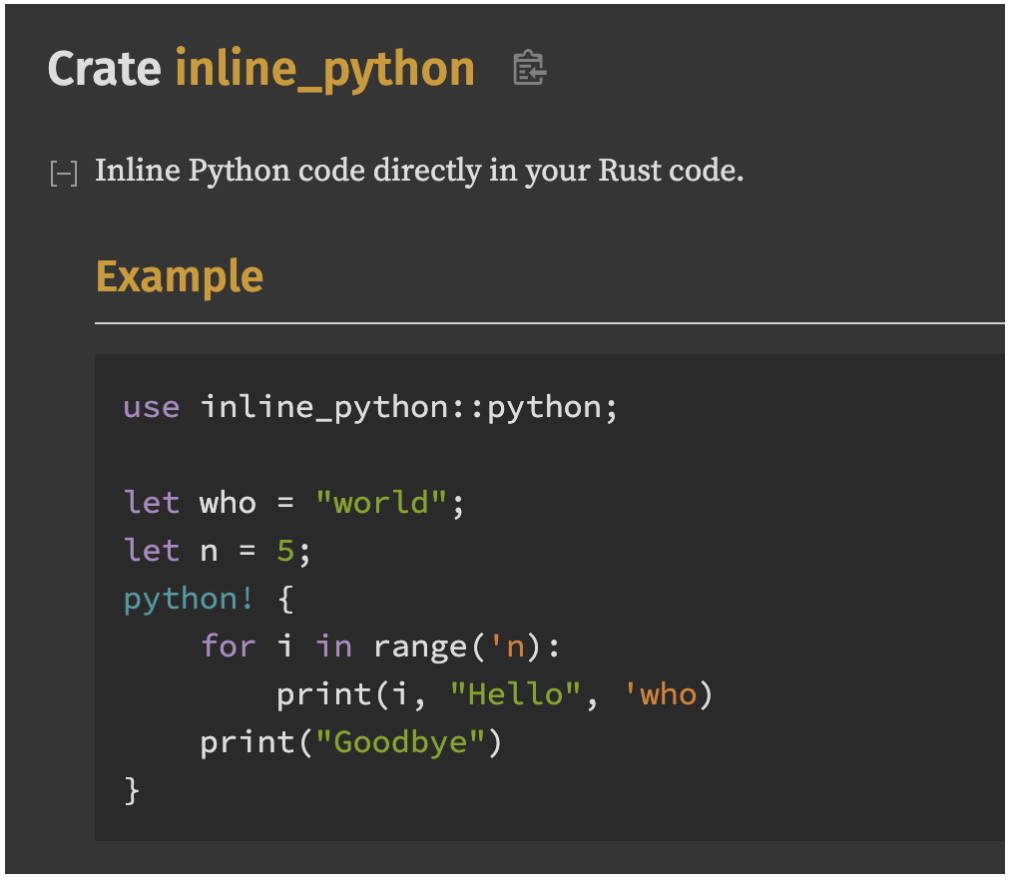


また、せっかく静的検査をやるのだからネイティブコードも出力したい

インタプリタ形式のPythonよりも高速化できることが期待できるし、シングルバイナリに固めて容易に配布できるというメリットもある

しかしPythonとの互換性を捨てたくはないので、バイナリとインタプリタの通信ができるようにしたい

※ Rustにはそういうバインディングライブラリがある



…と、いうことを実現する言語 Erg を作っています

これまでの進捗(6~9月)

言語機能

基本的な言語機能(e.g. 変数、関数、クラス)は既に実装済み

未踏期間中の主要なものとしては

- スライスの追加
- ユーザー定義再帰型の追加
- パターンマッチの機能強化

Language Server

未踏期間前から基本的な機能は実装済み

- [x] Completion
- [x] Diagnostics
- [x] Hover
- [x] Go to definition
- [x] Find references
- [x] Renaming
- [x] Inlay hint
- [x] Semantic tokens
- [x] Code actions
- [x] Code lens

Language Server

未踏期間中の進捗:

- Language Serverの並列化
- Language Serverのテスト基盤を開発・公開
  サーバーと通信を行うダミークライアントを実装、これを用いてテストも実装
  コンパイラ本体と分離できたので別リポジトリで公開
- パッケージ全体に対する検査・diagnostics報告
- (Language Server Protocolに関するWeb本を公開)

パッケージマネージャ

実用的なアプリケーションを作るにはパッケージマネージャが不可欠
パッケージマネージャはErg自身を用いて実装した

現在使用可能なコマンド:

- publish: 作成したパッケージを検証し、GitHub上で管理されるレジストリに登録
- build: コードをコンパイルして成果物をbuildディレクトリに置く
- clean: buildディレクトリをclean-upする
- help: ヘルプを表示する
- init: パッケージを初期化する
- run: アプリケーションパッケージを実行する
- install: (シェルスクリプトを使った擬似)実行可能ファイルを作って `$ERG_PATH/bin` に置く
- metadata: パッケージのメタデータを表示

ネイティブコードバックエンド

ネイティブコードバックエンドは、Rustコードをターゲットとする方式で実装

$$Ergスクリプト \xrightarrow{Ergコンパイラ} Rustコード \xrightarrow{Rustコンパイラ} バイナリ$$

ネイティブコードバックエンドは、構成的にはErg to Rustトランスパイラと Rustコンパイラを呼び出す部分からなる

トランスパイラ

トランスパイラの方は、現在

- 関数呼び出し(print, assert)
- 変数・関数・メソッド定義
- コントロールフロー(if, for, while, match)
- 基本的な二項演算
- クラス定義
- import(Ergモジュールのみ)

を変換可能

コンパイル(トランスパイル)実行例

```
~/Documents/GitHub/gal git:(main) (0.064s)
cat test.er

id x = x
add x, y = x + y

world = "world"
print! "Hello, {}! ", id world
print! "1 + 2 = {} ", add 1, 2
```

```
~/Documents/GitHub/gal git:(main) (0.811s)
cargo run -- compile test.er

    Finished dev [unoptimized + debuginfo] target(s) in 0.06s
     Running `target/debug/gal compile test.er`
```

```
~/Documents/GitHub/gal git:(main) (0.292s)
./test

Hello, world!
1 + 2 = 3
```

Rustに変換するので当然ではあるが、fibonacci関数での簡易ベンチマークではPythonよりも10倍以上高速に実行された

これからはErgとの互換性を高めるためにバインディングライブラリやPython APIを模倣するRustライブラリを作っていく

コンパイラ

- コンパイラ全体の並列化
- 型システムのバグ修正
- コード生成器のバグ修正

10月の進捗

パッケージマネージャ

- ロックファイル(npmのpackage-lock.jsonみたいなもの)の仕様を策定
  - ロックファイルの生成も実装
- 依存関係解決器を実装
  - 依存関係に基づいて再帰的にパッケージをダウンロードできるようになった

これでパッケージマネージャに最低限必要な機能は揃った

コンパイラ

並列コンパイルに関するバグの解決

- 7月より確認されていたものの大掛かりな改修となるため中々取り掛れなかったバグ
- Ergのモジュール(≒ファイル)は並列に解析されるが、これまではimport解決と意味解析を同一のフェーズで行っていた
  - 意味解析の途中でimportを見つけたら解析プロセスを立ち上げる
  - 本来は先にimport解決すべき、でなければデッドロックする恐れがある(実際これでパッケージマネージャのコンパイルがまれにフリーズしていた)

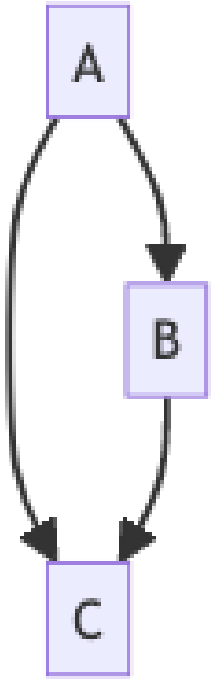
並列コンパイルに関するバグの解決

バグ発現例

以下のような状況を考える

モジュールCはモジュールA, Bから参照され、BはAから参照されている

Aが先にCのロックを取って解析終了を待つと、BはCのロックを得られず解析が進まない
AはBにも依存しているのでBの解析終了を待つ -> デッドロック！



並列コンパイルに関するバグの解決

解決策として、import解決を意味解析フェーズから分離し、先に行うこととした
意味解析の前に完全なパッケージの依存グラフを作成し、親モジュールのロックを取る権利を持つ子モジュールを一意に特定した

その他

- コード生成に関するバグを修正した
  - 即値のstore/loadコード生成に関するバグ
  - クロージャのコード生成に関するバグ
- 型推論のバグを修正した
- 型推論の順序を工夫することで推論能力を向上させた

今後の予定

- ネイティブコードバックエンドは、より多くのコードを変換できるよう機能強化を進めるほか、バインディング機構の実装を行う
  - パッケージマネージャをバイナリにコンパイルできるようにする
- フォーマッタの実装を進める