Go の資産を再利用できるコンパイラ基盤
— GGVM —

1. 背景

1.1. Go について

Go は、現在のソフトウェア産業界で最も勢いと需要のある言語の一つである。Go で実装されたソフトウェアや OSS ライブラリはコンテナ型仮想化の基盤ソフトウェアである Docker を始めとして、日に日に増えている。Go コンパイラの特徴としてソースコードを様々なアーキテクチャで動くシングルバイナリとしてクロスコンパイルできることが挙げられる。これはコンテナのサイズを小さくしたいモダンな Web アプリケーションアーキテクチャと相性がよく、実装言語として採用される理由に一役買っている。他にも

- コンパイラが生成したバイナリの実行速度のパフォーマンスが良い
- libc に頼らずとも syscall を呼べる様なとても豊富な標準ライブラリ
- goroutine による並行処理の簡潔さ
- 静的型付き言語流行に依る脱 LL（Ruby, Python）トレンド

が貢献していると考えられる。しかしその一方で、Go 自体は

- ジェネリクスを始めとした高級な型システムの欠如
- immutability
- 関数型言語にあるパターンマッチなどの高級な言語機能
- コード生成器が Go の AST と結合していて別のコンパイラフロントエンドから利用しにくい

など言語機能やコンパイラアーキテクチャの課題が挙げられる。

1.2. Java の資産を再利用している例

Java の流行以前は実行アーキテクチャによって書かれるコードが異なっていた。Java は内包している JVM によって、アーキテクチャを気にせずにコードを書けることを売りにした。豊富な標準ライブラリもまた流行に貢献しただろう。しかし、高級な言語機能を持つ言語が出てきたことによって Java に不満を持つプログラマが多くなった。そこで、Java より高級な言語をコンパイルして、JVM の命令列を生成できれば Java を書かずに資産を利用できるのでは？と実装されたのが Kotlin と Scala である。

特に Kotlin は Java で対応してる言語機能は全てサポートした上で、更に高級な言語かつ Java の資産をそのまま再利用できるので Java プログラマから絶大な支持を得ている。

余談だが、Go の並行処理機構である goroutine を Java が採用する Draft が現在ある。これをきっかけに goroutine のような並行処理を簡潔に扱える言語や処理系はますます増えていくだろう。

1.3. 自作言語実装におけるネイティブバイナリの生成

一般的に、自作言語実装におけるネイティブバイナリの選択肢は少ない。実用的なところで言うと、

- LLVM を使う
- 自力で対象アーキテクチャのアセンブリを生成する

の 2 択である。Go コンパイラにおける LLVM の採用は Russ Cox 氏が言及していて、採用が見送られた。要約すると、「LLVM は巨大すぎて Go で使うなら開発からもっと前から離れていた」というくらい大きいようだ。

LLVM は手元の計算機でビルドをするとまず、数十分のビルド時間、数十 GB のディスクスペースを消費する。これは LLVM を使った言語処理系の開発の場合、必ず払うコストでかつ LLVM IR をコンパイルする際に様々な最適化が実行されるのでとても遅くなる。巨大でコンパイラのバックエンドとして採用を見送るのもうなずける。

余談だが、Go コンパイラ本体を手元の計算機でビルドした場合、初回のビルドでも 10 分程度でビルドが終わる。もし LLVM を採用していた場合、上記の数十分の時間+LLVM のコンパイル速度分だけの時間がかかる。

2. 目的

1.2 における Java のように将来的に Go より高級かつ、Go の資産を利用できる言語が実装されるのではないかと推測している。更に、1.3 でも話題にしたように自作言語実装における実用的なネイティブバイナリ生成の選択肢はとても少ない。巨大な LLVM を使うか、自分で対象アーキテクチャのアセンブリを理解して出力する、を対象アーキテクチャの数だけやるかだ。そこに第 3 の選択肢として、LLVM より軽量かつ、ネイティブバイナリに Go のランタイムと豊富な標準ライブラリの資産が利用できるコンパイラ基盤を開発することが加わる。

本プロジェクトで開発するコンパイラ基盤 GGVM は、Java における Kotlin、Scala の様な、とてつもなく野心的で未踏性のあるプロジェクトを再生産するためのプロジェクトである。

3. 開発の内容

図 1 に本プロジェクトで実装した GGVM のアーキテクチャ図を示す。利用者の入出力は Input/Binary で、内部で与えられた入力は矢印の上のオブジェクトになる。実装に用いた言語は Rust で、Go のバージョンは 1.17.0 で固定している。

3.1. Input

図 2 の BNF で定義された文字列を入力に持つ。現状は、1 つの関数をコンパイルできる状態である。

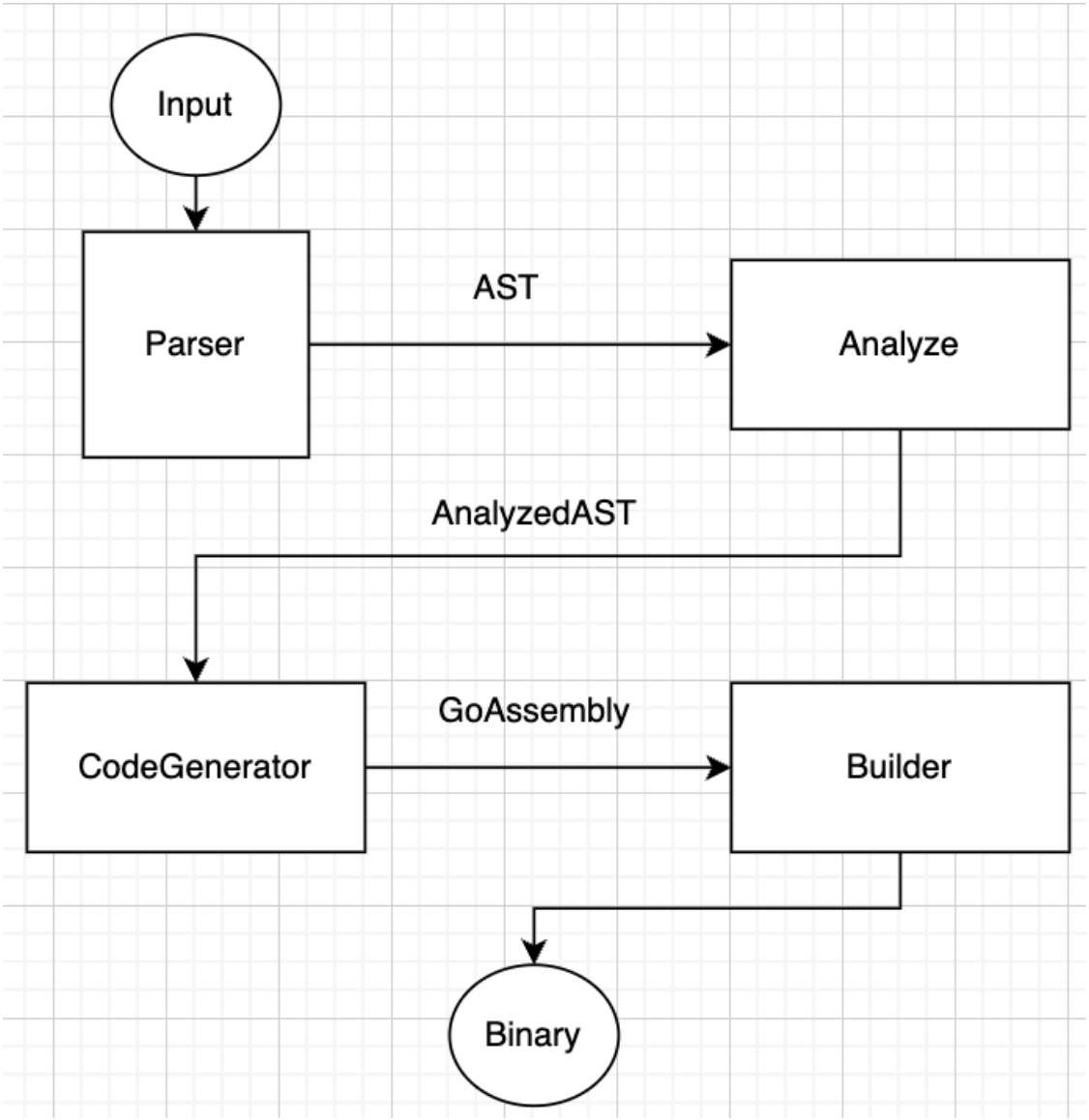

図 1：GGVM のアーキテクチャ図

3.2. Parser

Parser モジュールは GGVM-IR を AST オブジェクトに変換するために構文解析を行う。内部は nom というパーサコンビネータのライブラリを使用している。

3.3. Analyzer

Analyzer モジュールは AST と GoAssembly のギャップを埋めるために必要なモジュールである。一般的なコンパイラの意味解析を担当している。AST を解析して、

- スタックフレーム上の何番地に変数が格納されるか
- 受け取った AST

を結果のオブジェクトに埋め込む。

```
<Program> := <Func>
<Func> := func <FunctionName>() <Type> { <Statement> }
<Statement> := <LocalStatement> | <InstructionStatement>
<LocalStatement> := local <LocalIdent> <AddInstruction>
<InstructionStatement> := <AddInstruction> | <RetInstruction> | <CallInstruction>
<AddInstruction> := add <Type> <Operand> , <Operand>
<CallInstruction> := call <Callee>
<RetInstruction> := ret <Type> <Operand>
<Type> := int
<Operand> := <Var>
<Var> := %<Ident>
<FunctionName> := $<Ident>
<Callee> := <LocalIdent>
<LocalIdent> := %<Ident>
<Ident> := <Alpha>(<AlphaOrNumeric>)*
<AlphaOrNumeric> := <Alpha> | <Numeric>
<Alpha> := 'A' ... 'Z'
<Numeric> := '1' ... '9'
```

図 2：GGVM-IR の BNF

3.4. CodeGenerator

CodeGenerator モジュールは、解析された AST を元に Go Assembly へ変換する。

3.5. Builder

Builder モジュールは、生成された Go Assembly とブートストラップ用の main.go から Go コンパイラを用いてネイティブバイナリを生成する呼び出しを行う（図 3）。

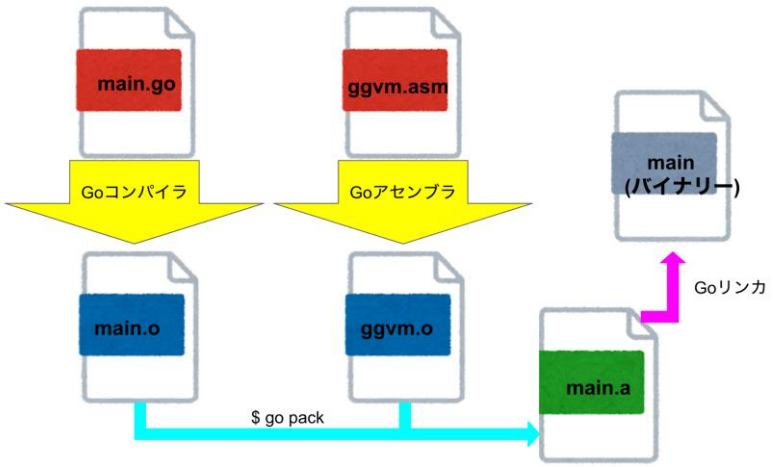

図 3：Builder がやっているビルドの図解

4. 従来の技術（または機能）との相違

GGVM は Go の資産を再利用することに重きを置いたコンパイラ基盤で、そのための機能を兼ね備えているのが LLVM や既存のコンパイラ基盤とは異なる点である。また LLVM は、任意のアーキテクチャをターゲットにできる汎用性を持つが、それ故にビルド時間やバイナ

リサイズなどが巨大になってしまう。GGVM のサポートは Go コンパイラがサポートしているアーキテクチャのみと割り切っており、その分コンパクトで扱いやすいコンパイラ基盤として選ばれるはずだ。

5. 期待される効果

ユーザが GGVM を利用することで、

- ネイティブバイナリを対象とした
- 豊富な Go の標準ライブラリを利用できる

自作言語を実装できるようになる。たいていの言語処理系のソースコードは処理系本体よりも標準ライブラリの量が多いので再実装の手間の省略に大きな効果が出るだろう。

6. 普及（または活用）の見通し

GGVM は現在、Go の資産を再利用できる状態ではありつつも実用的なコードを入出力できるわけではない。本プロジェクト期間終了後も開発を続けて完成度を高めていく。ある程度のフェーズに来たら GGVM を利用した言語処理系を実装する。Brainf**k のような簡単な言語から、最終的に Standard ML または OCaml と言った実用性がある言語を目指す。実装言語の選定理由としては Go のパラダイムとは異なる言語から Go の資産が再利用できれば、GGVM の価値をより強く裏付けられるためだ。また、入力である GGVM-IR の BNF はありつつも、利用しやすいラッパライブラリが未実装なので実装する。この実装が一通り終われば十分に普及・活用の見通しが立つ。

7. クリエータ名（所属）

Jantakorn Passawee（株式会社はてな）