多言語に対応可能なプロセス仮想マシン/コンパイラバックエンドの設計と実装

庄司　環

2024 年 1 月 16 日

1　Abstract

2　序論

2.1　動機

2.2　先行研究・プロジェクト

2.2.1　プロセス仮想マシン

■Java 仮想マシン　Java 仮想マシン（JVM）は、Oracle 社が開発しているプログラミング言語 Java のためのプロセス仮想マシンである。しかしながら、JVM 上で実行できる言語は Java のみではなく、Clojure や JetBrains 社による Kotlin などが実行できる。これは、JVM の仕様が公開されているため、JVM をターゲットとしたコンパイラを Oracle 以外も開発できるためである。#### .NET CLR #### BEAM #### Lua VM #### WebAssembly

3　用語の導入

本章では、本論文で用いられる用語、特に定義が研究者・技術者によって異なるものや、専門的で理解しにくいものについての導入を行う。## プログラムコンピュータの中枢である **CPU** は、機械語による命令列を解釈し、実行する。「機械語」というのはコンピュータのプロセッサが直接実行する言語の総称で、例えば通常の Windows PC に使われる x86_64 アーキテクチャの機械語、スマートフォンやタブレット、近年の Mac で使われる AArch64 アーキテクチャの機械語のように、機械語は CPU の **ISA**（命令セットアーキテクチャ）ごとに存在し、ISA が異なる CPU の機械語は互換性を持たない。また、機械語は 0 と 1 の列で構成されている。

プログラムとは、コンピュータに対する、記述された命令である。どのような形式であるかは問わない。たとえば、C や Python などのソースコードを指してプログラムと呼ぶこともあれば、Scratch のブロックの集合体もプログラムと呼ぶ。もちろん、機械語の命令列もプログラムである。## プログラミング言語 0 と 1 の列からなるという特性上、機械語は人には理解しがたい。そこで、人に理解しやすい言語でプログラムを書くことでプログラミングを容易にすることが考えられた。その言語が**プログラミング言語**である。プログラミング言語を実現する手法には大きく分けて二つの形式がある。一つは**コンパイラ**で、もう一つは**インタプリタ**で

ある。どちらもそれ自体がコンピュータ上で実行されるプログラムであるが、その内容は大きく異なる。コンパイラは、特定のルールに従って書かれたコードを機械語、もしくは別のプログラミング言語で書かれたコードに変換する。特に後者を**トランスパイラ**と呼ぶこともある。インタプリタは、それ自体が、プログラミング言語で書かれたコードを解釈し、実行する。

また、「プログラミング言語」と言ったとき、コンパイラやインタプリタなど、実際にソースコードを変換ないし実行するプログラムを指すことがある。「プログラミング言語が処理する前のソースコード」というのは、この用法である。また、コンパイラやインタプリタが入力とする「特定のルールに従って書かれたコード」の、「特定のルール」を指してプログラミング言語ということもある。簡単には、“Japanese” と言った時、「日本人」と「日本語」の両方を指すのに似ている。混同を防ぐため、本論文では、プログラムの方を**処理系**、このルールを**言語仕様**と呼称する。“Japanese” に例えると、日本語を理解する日本人が処理系で、構文や品詞や単語の意味など、日本語という言語そのもののルールを指すのが言語仕様だとなるだろう。## コンパイラ

4　仕様の定義

5　今後の展望

5.1　他言語関数呼び出しの簡易化

C 言語で書かれた、次のようなコードがあるとする。

```
// 設定ファイルを開いて、中身をパースして返す
 parseConfigFile(path: string) {
    config_file = file.open(path) //設定ファイルを開く

    // ... いくらかの処理 ...

    return parsedConfig
}
```

このプログラムは、あるフォーマットで書かれた設定ファイルへのパスを受け取り、そのファイルを開き、パースして使いやすくしたものを返す。ここで、パースする設定ファイルのフォーマットが非常に優れたものだったため、ほかのプロジェクトでも使いたいと思ったとしよう。そのプロジェクトでは、B という別のプログラミング言語を使っていた。`parseConfigFile` という既存の資源があるので、DRY の観点からそれを用いたい。しかしながら、次のような B のコードを書いても関数 `parseConfigFile` を用いることはできない。

```
func load_config() {
    import config

    const path_to_file = "C:/Users/username/.appconfig"
    const parsed_config = config.parseConfigFile(path_to_file)
}
```

これはプログラミングをする人から見ると当然だろう。実際に B で `parseConfigFile` を用いるためには、例えば B が Rust 風の他の言語で書かれた関数を呼び出す仕組みを持っているとすると、次のようなコードを書く必要があるだろう。

```
func load_config() {
    extern {
        func parseConfigFile(path: a_string) -> a_string
    }
    const path_to_file = "C:/Users/username/.appconfig"
    const parsed_config = parseConfigFile(a_string.new(path_to_file))
}
```

このコードの `extern` ブロック内では、B 処理系に `parseConfigFile` の型を示している。引数## 仕様の形式的な定義 Grond 上で用いられる言語の意味は自然言語で定義されている。しかしながら、将来的には表示的意味論や操作的意味論などの形式的意味論での定義も行う、もしくは完全に形式的意味論での定義に切り替える必要がある。「プログラミング言語の形式的意味論入門」では、形式的意味論を与えることをプログラミング言語に数学的なモデルを与えることだとしており、そのメリットについて次のように述べている。>これによって，プログラムの挙動に関する理解や推論の基礎づけが可能となる．数学的なモデルは，様々な解析や検証に有用なだけではなく，さらに基礎的なレベルでも役に立つ.

また、同書のカバーそでには、次のように述べられている（掲載箇所から、訳者や編集者による文である可能性が高いだろう）。>プログラムを数学の俎上に載せることにより、プログラムやプログラミング言語を厳密に理解し、解析し、これらについて推論することができる。

このように、形式的意味論を与えることでプログラムを数学的に解析することが容易になる。それは、安全性などの面で非常に重要である。## コンパイラの導出 コンパイラバックエンドは Grond Backend で提供されるが、フロントエンドはプログラミング言語の開発者が自ら実装する必要がある。フロントエンドのうち、レキサやパーサは `lex` や `yacc` など、すでに自動生成するプログラムが存在している。しかし、抽象構文木から中間表現を生成するプログラムの例は極めて少ない。しかしながら、操作的意味論を用いたり、どのような GIR に変換するのかを定義するなどして意味論を与えることで、中間表現に翻訳するプログラムは恐らく実現可能である。もしもそのようなプログラムを実現できたなら、パーサジェネレータやレキサジェネレータなどと併せることで、文法と意味論の定義ファイルからフロントエンドを導出することが可能になる。そうすると、バックエンドに Grond Backend を用いることで、コンパイラの導出が可能になる。コンパイラの開発には大きな労力が必要だが、これが実現できればそれを大きく減らすことができる。## 互換レイヤとしての利用一般的に、OS や CPU のアーキテクチャが異なれば、実行ファイルに互換性はない。これは、例えばシステムコールが異なったり、機械語が異なったりするからである。しかし、互換性のない環境のプログラムを動かすためのソフトウェアが存在している。それらはシステムライブラリを模倣したり、システムコールを変換したり、互換性のない機械語のコードをその環境で実行可能なコードに変換したりすることによって実現される。それぞれ、Windows 向けアプリケーションを UNIX で動作させる Wine、FreeBSD で Linux バイナリを動かす互換機能（Linuxulator）、PowerPC 向けバイナリを x86 で動作するように、また x86_64 向けバイナリを AArch64（Apple Silicon）で動作するようにする Mac OS X/macOS の Rosetta/Rosetta 2 が例である。カーネルを実行するオーバーヘッドがないなどの理由から、仮想マシンなどで実行する場合と比べて、互換レイヤを用いることは性能上のメリットがある。しかし、OS ごと完全に実行する仮想マシンでは、プロ

グラムは基本的には物理マシンと同様に動作するが、互換レイヤでは対応するシステムライブラリが未実装であるなどの理由から、動作しないプログラムも存在する。ところで、Grond はコンポーネント同士の結合を疎にするよう設計されている。例えば、`APICall` に対応する処理を呼び出すコンポーネントは、Grond に依存しないライブラリとして用いることも可能である。そこで、例えば x86_64 の Linux バイナリを AArch64 の macOS で動作させる互換レイヤを書くとき、# 付録

5.2 参考文献

通常、論文では Web サイトを参照することは避けるべきだとされている。しかし、本論文では先行研究として挙げたプロジェクトの公式リファレンスや、公式のソースコードリポジトリが Web 上に存在する。この場合においては、むしろ第三者による書籍よりもこちらの方が信頼性は高いと考えられるため、これに限って本論文では Web サイトへの参照を避けない。

5.3 実装