卒業論文 2015 年度 (平成 27 年)

Bootstrap に向けた Swift による Swift 構文解析器の設計と実装

慶應義塾大学 環境情報学部
出水 厚輝

卒業論文要旨 - 2015 年度 (平成 27 年度)

Bootstrap に向けた Swift による Swift 構文解析器の設計と実装

現在利用されている多くの高級な汎用プログラミング言語では、コンパイル対象となる言語自体でそのコンパイラを記述する Bootstrap が行われている。Bootstrap を行うことによるメリットはいくつかあるが、度々モチベーションとしてあげられるのは、現存するプログラミング言語よりも Bootstrap を行おうと考えているプログラミング言語のほうが後発のものであるため、より表現力が高く開発しやすいという点である。

しかし、近年開発されている汎用プログラミング言語に至っては、その言語自体だけでなく最初にコンパイラを記述する言語も高級なものとなっており、対象のコンパイラを記述する上でどちらの方がより高い表現力や性能を持つかを簡単に判断することはできなくなってきている。

Apple 社が中心となって開発しているプログラミング言語 Swift もそのメリットとデメリットを明確に評価することができず、Bootstrap するべきか否かの判断を下せていない汎用プログラミング言語の 1 つである。現在最も有名な Swift のコンパイラ実装は C++ で記述されており、コンパイラの核となる構文解析においても C++の特徴的な機能を駆使して、より低級な言語ではボイラープレートとなるコードを排除している。Swift はその可読性の高さと実行速度の速さを謳った言語であるが、その性能が Swift コンパイラという大規模なソフトウェアにおいて C++を相手としても通用するものであるかどうかを形式的に議論することは容易ではない。

そこで本研究では、Swift で記述した Swift の構文解析器を実装し、その実行時間とソースコードの行数を現行の Swift コンパイラ中の構文解析器と比較することで、Swift が Bootstrap を行うための判断材料を収集・考察する。本論文では、Swift で構文解析器を書き換えることによって可読性につながりうるソースコードの行数の削減は実現できるが、実行速度の面においては未だ Swift 自体が充分な性能を持っていない可能性があることを示し、その結果から Swift が Bootstrap を行うならば必要になるであろうステップについて考察を行っている。

キーワード:
1. コンパイラ・ブートストラップ, 2. 構文解析, 3. 構文解析器の実装,
4. プログラミング言語 Swift

慶應義塾大学 環境情報学部

出水 厚輝

Abstract of Bachelor's Thesis - Academic Year 2015

Design and Implementation of Swift Parser Written in Swift for Bootstrapping

English abstract here.

Keywords :
1. Bootstrap a Compiler, 2. Parser, 3. Implementation of Parser,
4. Swift Programming Language

Keio University, Faculty of Environment and Information Studies

Atsuki Demizu

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

2015年12月4日、Apple社が予てより同社の提供するCocoaおよびCocoa Touchフレームワークを用いたソフトウェアの開発用として提供していたプログラミング言語Swiftをオープンソース化し、Linuxを中心としたさまざまなプラットフォームにおけるソフトウェアを開発するための拡張を開始した。これによりプログラミング言語SwiftはObjective-Cの担ってきたiOSやMac OS Xなどにおけるソフトウェアの開発だけでなく、C++やJavaなど他の汎用プログラミング言語が担ってきたソフトウェア開発においても、それらの代替となり得る可能性を持つこととなった。

Swiftはオブジェクト指向や全称型・存在型の導入、関数の第一級オブジェクト化、HindlyとMilnerによる型再構築アルゴリズムの採用など、現在多くのプログラマに使用されている他の汎用高級言語が持つ様々な特徴を持っているが、まだその特徴を採用するか否かがよく議論されていないものもある。その内の1つがコンパイラをそのコンパイル対象の言語自体で開発するBootstrapプロセスの採用である。

表1.1はWeb検索エンジンにおけるクエリヒット数からプログラミング言語の知名度を格付けしたTIOBE Indexの2015年12月版において上げられている言語の内、汎用言語であるものだけを上位から20言語抽出し、それらの主要なコンパイラがその言語自体で記述されているかを示したものである。この20言語の内だけでもBootstrapを行っているものが8言語あり、その中に性能の問題からコンパイラ用の言語として採用されづらいインタプリタ型言語なども含まれていることを考慮すれば、かなりの言語がBootstrapされていることが分かる。しかし、現在SwiftはC++を用いて開発されており、Bootstrapは行われていない。

現在Swiftにおいては未だ多くの機能が不足しており、他の問題を優先しているためにBootstrapについて大きく取り上げられてはいない。また、開発者のメーリングリスト [1]では特にSwiftコンパイラのバックエンドとして採用されているLLVMのAPIがC++で提供されていることから、SwiftがC++と同様の役割を果たすにはもう少し時間が必要だという意見も上がっている。

しかし、2.1節で述べるようにBootstrapを行うことで得られる利益があることが他の言語の事例によって示されている以上、十分な議論なしに現状のSwiftにBootstrapが不要であると判断するのは早計であるといえる。

表 1.1: 知名度の高いプログラミング言語の Bootstrap 状況

順位	言語名	コンパイラ名	Bootstrap 状況
1	Java	javac	N (C, C++?)
1	Java	OpenJDK	N (C++, Java)
2	C	gcc	N (C++)
2	C	clang	N (C++)
3	C++	gcc	Y
3	C++	clang	Y
3	C++	Microsoft Visual C++	Y
4	Python	cpython	N (C)
4	Python	PyPy	Y
5	C#	Microsoft Visual C#	N (C++)
5	C#	.NET Compiler Platform (Roslyn)	Y
6	PHP	Zend Engine	N (C)
7	Visual Basic .NET	Visual Studio	N (C++, C#)
7	Visual Basic .NET	.NET Compiler Platform (Roslyn)	Y
8	JavaScript	SpiderMonkey	N (C, C++)
8	JavaScript	V8	N (C++, JavaScript)
9	Perl	perl	N (C)
10	Ruby	Ruby MRI	N (C)
12	Visual Basic	Visual Studio	N (C++, C#)
13	Delphi/Object Pascal	Delphi	N (?)
13	Delphi/Object Pascal	Free Pascal	Y
14	Swift	swift	N (C++)
15	Objective-C	clang	N (C++)
15	Objective-C	gcc	N (C++)
17	Pascal	Free Pascal	Y
17	Pascal	GNU Pascal	N (C, Pascal)
20	COBOL	GnuCOBOL	N (C, C++)
21	Ada	GNAT	Y
22	Fortran	GNU Fortran	N (C, C++)
22	Fortran	Absoft Fortran Compiler	?
23	D	DMD	Y
24	Groovy	groovy	N (Java, Groovy)

1.2 本研究が着目する課題

Swift コンパイラが抱える他の問題との優先度や使用しているフレームワークとのつなぎ込みに関する問題が解決したとしても、Swift コンパイラの Bootstrap を行うかどうかという判断を下すにはより根源的な課題がある。それは、現在 Swift を記述している C++ 言語が Swift と比較しても高い表現力を持っているために、 2.2 節で見る幾つかの事例とは異なり、Bootstrap を行うことで得られるメリットや、そもそも現行のコンパイラで使用されている手法を維持したまま Bootstrap を行うことが可能であるか否かが自明でないというものである。

また、現在の Swift は C++との相互運用性を持っていないため、コンパイラ中の一部分を Swift で記述したものに置き換えることは難しく、逆に実際に使用されているコンパイラ中のモジュール化が可能なほど大きなパーツを Swift へ移植するとなると、その間の言語への機能追加などの改変は現行のものと移植中のものの両者に適用するか、移植中のもののみに追加して移植が完了するまでその適用を先送りしなくてはならなくなってしまう。

1.3 本研究の目的

本研究では、Swift コンパイラが Bootstrap することによって得られるメリットと被るデメリットを定量的に示し、Bootstrap を行うべきか否かを判断する上で有用な情報を収集することを目的とする。そのためのアプローチとして、Swift コンパイラの基幹的機能である構文解析器を Swift によって実装し、その実行時間とソースコードの行数を現行の Swift コンパイラの構文解析器と比較する。また、この独自の構文解析器は現行の Swift コンパイラと基本的な設計手法において同じものを採用するだけで、完全に独立させたものとして実装する。

この方法により、現在の Swift コンパイラの開発状況などの影響を一切受けずに Bootstrap のための評価が可能となり、またその評価が Bootstrap の可能性に対して有意義な知見を与えることを提示する。

1.4 本論文の構成

本論文の構成は次の通りである。

第 2 章では本研究の考察対象であるコンパイラの Bootstrap についてそのメリットについてまとめ、Swift 以外の言語における事例と課題について整理する。第 3 章では本研究が着目するプログラミング言語 Swift の特徴とそのコンパイラ実装の基幹部分における特徴について説明する。第 4 章では現行の Swift コンパイラとの比較対象となる Swift で記述した Swift コンパイラ「TreeSwift」の構成について述べ、現行のコンパイラとその基本的な設計手法などに大きな差異がないことを確認する。第 5 章では現行の Swift コンパイ

ラと TreeSwift の構文解析器についてその実行速度とソースコードの行数を比較し、その結果について考察する。第 6 章では本研究の結論と今後の展望についてまとめる。

第2章　コンパイラのBootstrap

2.1　Bootstrapの利点

2.2　Bootstrapの事例

2.3　Bootstrapの課題

第3章　プログラミング言語Swift

3.1　Swiftの目的

3.2　Swiftコンパイラの構成

3.3　Swiftコンパイラの基幹的機能

3.4　Swiftコンパイラの課題

第4章　TreeSwiftの設計と実装

4.1　実装の概要

4.2　構文解析

4.3　その他の実装

第5章　評価

5.1　評価概要

5.2　構文解析における性能差

5.3　ソースコードの比較

5.4　考察

第6章　結論

6.1　本研究のまとめ

6.2　今後の展望

謝辞

参考文献

[1] [swift-dev] Bootstrapping Swift compiler. https://lists.swift.org/pipermail/swift-dev/2015-December/000004.html, December 2015.