スクリプト言語 KonohaScript の LLVM を用いた高速動作可能な実行環境の設計と実装

井出真広[†] 若松悠樹[††]
志田駿介[††] 倉光君郎[†,†††]

Perl, Ruby, Python などスクリプト言語は従来，柔軟性・記述性の高さを重点に設計されてきたが，適応領域が広がるにつれてパフォーマンスが重要となってきている．本稿は静的型付けスクリプト言語 KonohaScript を対象に，コンパイラ基盤である Low Level Virtual Machine(LLVM) を用いた KonohaScript コンパイラのバックエンド (LLVM バックエンド) の設計，実装について述べる．評価では，インタプリタと比較し高速化が実現できたことを示す．

The implementation of Just in time compiler for KonohaScript using LLVM.

Masahiro Ide ,[†] Yuki Wakamatsu ,[††] Syunsuke Shida [††]
and Kimio Kuramitsu [†,†††]

Recently, scripting language has been used some important situations and it has been required runtime performance. To improve performance, many scripting language use Just-In-Time (JIT) compilers.

In this paper, we desigin and implements a new backend of a KonohaScript compiler that leverage Low Level Virtual Machine(LLVM). we discuss differences between LLVM and KonohaScript design. Our evaluation shows that this design obtains improvement in performance compared to the current implementation.

1. はじめに

従来，スクリプティング言語は，その柔軟性，拡張性の高さから，システムコンフィギュレーション用途などに使われ，実行時の処理性能よりもプログラミングのしやすさを重視して言語設計が行われてきた．しかし，近年スクリプティング言語は Web アプリケーション，ゲーム開発など適応領域が広がる傾向にある．その結果，従来重視されていなかった処理速度が重要となってきている．そのため，それぞれの言語処理系ではバイトコード評価器や，ネイティブコードの動的な生成を行う Just-in-time (JIT) コンパイラの導入により言語処理性能を向上させている．

我々は，KonohaScript[9)8)] と名付けた静的型付けによるスクリプト言語の開発を行っている．KonohaScript は，コンパイル時型検査や型情報にもとづく最適なバイドコード生成とバイトコード評価器により他のスクリプト言語処理系と比較して，高い処理速度を実現している．また，JIT コンパイラによるネイティブコード生成を実験的に実装している[7)] が，V8[6)] や TraceMonkey[5)] などのプログラム言語処理系と比較すると性能が劣る．

本研究では KonohaScript におけるネイティブコード生成器の生成するコードの実行効率を改善し，性能向上を目指す．

我々は，目的の達成のため，KonohaScript コンパイラのバックエンドとして Low Level Virtual Machine(LLVM)[10)] を用いた LLVM バックエンドの設計を行う．LLVM は動的なコード生成，プログラム実行時のコード解析，最適化をサポートしたコンパイラフレームワークである．

本稿では，KonohaScript コンパイラの上に LLVM バックエンドの設計と実装を行い，得られた知見について述べる．そして，提案する LLVM バックエンド

† 横浜国立大学大学院
Graduate School of Yokohama National University
†† 横浜国立大学
Yokohama National University
††† 日本科学技術振興機構 CREST
Japan Science and Technology Agency/CREST

の機能と性能をベンチマークアプリケーションを用いて評価を行う．ベンチマークによる評価では，インタプリタと比較し平均で約 3 倍の高速化が実現できたことを示す．

本論文の構成は, 次のとおりである．第 2 節では，KonohaScript における既存のコード生成系の設計について述べ，第 3 節にて提案する LLVM バックエンドの設計と実装を理解する上で必要となる箇所について述べる．第 4 節で LLVM バックエンドについての設計，第 5 節で実装を詳しく述べる．第 6 節では性能を評価し，第 7 節にて関連研究を述べ，第 8 節にてまとめと今後の課題を述べる．

2. KonohaScript 言語処理系の構造

KonohaScript では 2 つのコード生成器を持つ．1 つ目はバイトコード生成器であり，VM で実行可能なコードを生成する．2 つ目はネイティブコード生成器であり，プログラム実行中にバイトコードからネイティブコードを生成する JIT コンパイラを持つ．以下ではそれぞれの動作について説明する．

2.1 KonohaScript のバイトコード

KonohaScript ではスクリプトは Konoha コンパイラによって型付けされた抽象構文木 (AST) に変換される．そして，生成された AST は KonohaScript 専用のバイトコードへコンパイルされ，KonohaVM 上で実行される．**図 1** は整数の加算をおこなう KonohaScript プログラムである．KonohaScript コンパイラは，このプログラムから図 1(2) に示すバイトコードを生成する．

次に，KonohaScript のバイトコードの定義の一部を**表 1** に示す．表 1 では各バイトコードのオペランドとバイトコードの動作を示している．VM レジスタを第 1 オペランドにもつバイトコードは，対象の VM レジスタへ代入操作があることを表し，第 2 オペランド以降に VM レジスタ名をもつバイトコードは，その VM レジスタを参照することを表している．

2.2 Konoha コンパイラ

本節では図 1 の例を用いて，Konoha スクリプトがどのような単位でバイトコードにコンパイルされるか，また KonohaVM がバイトコードを実行する部分についてを述べる．

図 1 は KonohaScript スクリプトの記述例である．Konoha コンパイラはこのスクリプトを受け取り，バイトコードコンパイルを行う．Konoha コンパイラはスクリプトのトップレベルのコードも一旦メソッドとしてコンパイルするため，メソッドを 1 つのコンパイル単位としてバイトコードを生成する．図 1 に示す KonohaScript スクリプトからは トップレベルのコード (図 1 (1)) とメソッド **f** (図 1 (2)) の 2 つのメソッドのバイトコードが生成される．図 l の (1)，(2) の各行は左から順にバイトコード上のラベル，バイトコード名，バイトコードを実行する際のオペランドを示す．

命令コード	オペランド	説明
NSET	r_1 3	Puts the integer constant into r_1
iADD	r_2 r_3 r_4	r_2 = r_3 + r_4 and puts the result into r_2
OMOV	r_4 r_2	Move the object reference in r_2 to r_4
XNMOV	r_0 fieldID r_2	Puts r_2 into a field.
CALL	r_2 r_{10} r_{16} method	call method
IJLTC	L_4 r_3 5	Jumps to L_4 if r_3 < 5
JMP	L_1	Jumps to L_1
RET	-	return

表 1 バイトコードの定義の一部

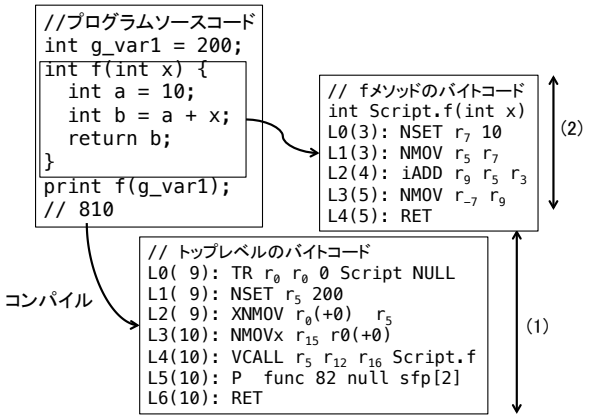

図 1 KonohaScript とバイトコード例

2.3 JIT コンパイラ

KonohaScript では JIT コンパイラによってバイトコードをプログラム実行中に機械語に変換することが可能である．JIT コンパイラでは，バイトコード評価器の機械語をコードテンプレートとして利用仮想マシンの命令列を器械後に変換する．

一般に，JIT コンパイラは VM 上で実行されるコードと比較して実行効率の高いコードを生成することが可能となる．しかし，対象となるアーキテクチャに関する知識と作業量を要求するため，開発コストが非常に高くなる．現在の実装では x86, x64 のみをサポートしており，他のアーキテクチャはサポートしていない．

3. LLVM の利用

本節では LLVM の中間表現，型情報について以降の節を説明するために必要な情報について述べる．

3.1 LLVM IR

LLVM の中間表現 LLVM IR は LLVM をコード生成器として利用する際の入力としてプログラムを記述するための言語である．しかし，LLVM IR は LLVM の内部表現としてもプログラム解析パス，最適化パス中でも利用されている．LLVM IR はアセンブリ言語に似た低級言語であるが，プログラム中に静的な情報として型情報やデータフロー情報を保持している．特にデータフロー情報は static single assignment(SSA) 形式[2] を用いてプログラム中に表現している．SSA 形式とは，プログラム中で使用される各変数への代入 (変数の再定義) が，プログラム中の一箇所で行われるような中間表現の形式である． SSA 形式を用いた最適化 (SSA 最適化) は，データフロー解析と比べて，大域的な最適化をより効率的に行うことができる．

LLVM IR は静的な情報と低級言語を組み合わせることで効率的に様々なプログラム言語をサポートすることが可能で，また積極的な最適化を行うのに十分な言語であると言える．LLVM IR の主な特徴は以下のとおりである．

- 型情報をもったアセンブリ言語
- ハードウェアのレジスタを抽象化し，プログラム中ではレジスタを制限なく利用可能
- SSA 形式, ϕ 関数を用いてプログラムを記述
- 基本ブロック (BasicBlock) と明示的な分岐命令によって制御フローを記述
- メモリアクセスには型安全なメモリアクセス命令 `getelementptr` 命令を持つ

SSA 形式では，プログラム中のすべての代入が単一になるようにするため，制御フローグラフにおける制御の合流するブロックの先頭に ϕ 関数を導入し，それ以降の変数の使用を ϕ 関数で定義した変数と対応させる．**図 2** は LLVM IR で記述した階乗を求める関数の例である．LLVM IR で記述した関数は基本ブロックのリストで表現されている．fact 関数は 3 つの基本ブロックで構成されており，それぞれ EntryBlock, return, call とラベル付けされている．各変数はそれぞれ別々の名前が割り当てられており，仮想レジスタ上に割り当てられる．また，すべての操作には型情報 (例えば，i64 はビット長が 64 の整数型を示す．) が付与されている．

```
define i64 @fact(i64 %n) {
EntryBlock:
  %cond = icmp sle i64 %n, 1
  br i1 %cond, label %return, label %call
return:
  ret i64 1
call:
  %arg = sub i64 %n, 1
  %x = tail call i64 @fact(i64 %arg)
  %res = mul i64 %n, %x
  ret i64 %res
}
```

図 2 n の階乗を求める LLVM IR コード

4. LLVM バックエンドの設計

我々は, LLVM を用いた KonohaScript の積極的な最適化を目的とし，KonohaScript で記述されたプログラムから LLVM IR へ変換する LLVM バックエンドの設計を行った．本節では，コード生成器に LLVM を用いる目的について述べたのち，提案する LLVM バックエンドの設計について述べる．

4.1 LLVM によるコード生成

前節までで述べたとおり，KonohaScript では AST からバイトコードを生成するバイトコード生成器とバイトコードからネイティブコードを実行時に生成する JIT コンパイラを持つ．バイトコード生成器の利点はインタプリタで実行可能なコードを生成することで可搬性に着目している点である．また，JIT コンパイラの利点は可搬性は低いものの高い処理速度を実現できる点である．

我々は，LLVM を KonohaScript コンパイラのバックエンドとして統合することで可搬性と実行効率の高い実行処理系を実現できると考えた．バイトコード生成器，JIT コンパイラと比較して LLVM バックエンドには以下の利点がある．

- コード生成器の分離
- 最適化パスの利用

まず，コード生成器の分離については, 効率の良いコードを生成する生成器を構築するためには開発コストが高く，また生成されたコードが安定的動作する環境を提供する必要がある．

LLVM はコンパイラフレームワークとして 10 以上のアーキテクチャに対して C/C++コンパイラや他の言語処理系のコード生成器として利用されている点から LLVM を用いることで既存の 2 つのコード生成器と比べより低い開発コスト，メンテナンスコストで効率的なコードを安定的に提供することができると考え

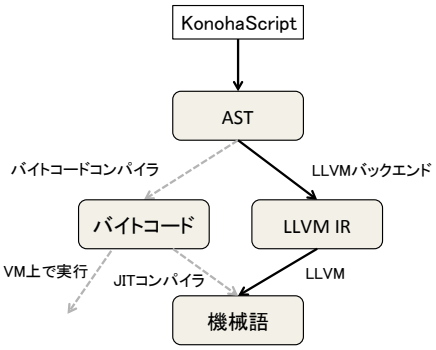

図 3 KonohaScript LLVM バックエンド処理フロー

られる．

また，最適化パスについては，現在 KonohaScript ではバイトコード生成器，JIT コンパイラの両者でピープホール最適化，メソッドの inlining 最適化を提供している．LLVM では更にループ最適化，部分冗長式の除去など積極的な最適化を行うことが可能なため，より効率的なコード生成が実現可能だと考えられる．

4.2 LLVM バックエンドの開発方針

我々はまず，KonohaScript のコード生成器として LLVM を利用するにあたり，以下の 2 つのコード生成方式について検討を行った．

- 型付けされた AST から LLVM IR の生成を行う
- バイトコードから LLVM IR の生成を行う

KonohaScript は静的型付けであり，整数や浮動小数点数などの型を LLVM IR 上でも同様に表現できるため, いずれの場合も効率的なコード生成器の実装が可能であると考えた．本論文では，LLVM IR 上での KonohaScript の積極的な最適化を目的とするため，制御フロー情報などプログラムの情報がより多く残っている AST から LLVM IR を生成する方式，LLVM バックエンドを導入する形で設計を行うこととした．LLVM IR を生成する手順を**図 3** に示す．

4.3 KonohaAST と LLVM の比較

LLVM IR では，多くのプログラム言語，アーキテクチャに対応するために LLVM IR 上で表現できる変数の型，関数呼び出しの仕組みが定義されている．一方で KonohaScript の AST(以降 KonohaAST とする) は KonohaScript をバイトコードに変換するため専用に設計されており，KonohaScript のプログラムを LLVM IR を用いて記述するためには KonohaAST と LLVM IR との間でデータの変換を行う必要がある．本節では以下の項目について LLVM IR と KonohaAST の比較を行い，それぞれの対応と変換方法について述べる．

- 型の定義
- 変数, 制御フロー情報
- 関数呼び出し

4.3.1 型 の 定 義

LLVM IR では多くのプログラム言語のサポートを目指すために，bool 型 (i1 型), char(i8 型), long(i32 型) など N ビット長の整数型をサポートしており，また複数の浮動小数点型を提供している．また，構造体や配列のサポートも行なっている．

```
i1, i2, …, i32, … iN
float, double, fp80
%array = type [3 x i64] /* 配列 */
%struct = type { i64, float } /* 構造体 */
```

KonohaScript ではプリミティブ型として boolean, int, float の 3 種類を用意している．データ構造を扱う際に KonohaScript は配列，クラスを用意している．すべてのオブジェクトは GC の実装を平易化するために各オブジェクトの大きさを 64byte に揃えており，以下のメモリレイアウトを持つ．

```
typedef struct {
  knh_hObject_t h; // オブジェクトのヘッダ情報
  struct Object  **fields; //クラスフィールド
} knh_Object_t;
```

構造体，オブジェクトのフィールドに保持されている値のロード処理を考えた時に，LLVM IR では `getelementptr` 命令を用いて 1 回のメモリアクセスで値を取得できるのに対し，KonohaScript では一度フィールドの先頭アドレスを取得した後，実際のフィールドの値をロードする形を取る．

KonohaScript のランタイムライブラリやオブジェクトのフィールドへアクセスを行う際に LLVM IR を用いて表現できるように KonohaScript のランタイムライブラリで定義されている構造体, 型について LLVM IR を用いて構築し，LLVM の最適化によって最適化可能な形に変換を行った．

4.3.2 変数，制御フロー情報

次に，KonohaAST と LLVM IR の変数，制御フローの扱いについて比較を行う．KonohaAST では LLVM IR と同様にすべての変数は静的に型付けされており，また変数は個数の制限なく利用することが可能となっている．しかし，KonohaAST は SSA 形式ではないため，変数の宣言と利用が一意に決めることができない点が LLVM IR と異なる．

KonohaAST の変数と LLVM IR で表現した変数の対応表を各基本ブロック毎に用意し，ある変数について各先行ブロックが変数の再定義を行った際に，制御

の合流点で ϕ 関数を挿入し，変数の定義が唯一となるように対応した．

4.3.3 関数呼び出し

最後に，LLVM IR と KonohaScript の関数呼び出し規約について比較を行う．LLVM IR ではアーキテクチャの呼び出し規約に沿って関数呼び出しを行う．(例えば Intel アーキテクチャの場合には stdcall や fastcall など)

一方，KonohaScript ではすべてのメソッドは以下の専用のインタフェース `knh_Fmethod` 型に合わせた形でメソッドが定義される．

```
void (*knh_Fmethod)(CTX ctx,
          knh_sfp_t *sfp, long _rix);
```

ここで，ctx は言語ランタイムの管理情報 (Context) であり，`sfp` は KonohaScript ランタイムのスタックポインタ (Konoha スタック)，`_rix` は返り値が置かれるスタック上のインデックスを示している．これに合致する C の関数であれば，API 定義スクリプトに書かれたメソッドから呼び出すことができる．メソッドを呼び出す際，呼び出し元の関数はスタックポインタ上に引数を push し，関数を呼び出す．

KonohaScript の変数はプログラム実行時には VM の仮想レジスタ上に配置され，メソッドを呼び出す際はスタックに引数を配置し VM のメソッド呼出命令によってメソッドの呼び出しが行われる．既存の呼出インタフェースを変更せずに KonohaScript を LLVM IR に変換するために LLVM バックエンドでは引数，返り値については直接 LLVM IR で表現せず，KonohaScript のスタック `sfp` へのメモリアクセスという形で表現することとした．

また，KonohaScript は正確な GC を採用しているため，LLVM IR で表現された生きているオブジェクトを誤って GC が回収されるのを防ぐ必要がある．GC が発生する可能性のある操作 (メソッド呼出，ランタイムライブラリ呼出) を行う前に一旦すべての変数をスタックに退避させ，その後メソッド呼び出しを行うようにコード生成を行うよう対応を行った．

5. 実行速度向上の工夫

本章では LLVM バックエンド に適用した実行速度向上のための工夫について述べる．

5.1 特 化 命 令

KonohaScript では加算，減算など数値演算も含めた全ての操作はメソッド呼び出しの形で表現されるが，バイトコード上では数値演算処理や変数のキャストなどの操作については特化命令が用意されている．LLVM IR では基本命令として整数演算，浮動小数点演算や制御命令が用意されているため，LLVM IR で表現可能な KonohaScript のメソッド呼び出しについて変換を行った．

5.2 メソッド呼出の工夫

第 4.3.3 節で述べたとおり KonohaScript のメソッドは共通のインタフェース `knh_Fmethod` 型に合わせた形で定義される．そのため LLVM IR で記述されたメソッド同士でメソッドを呼び出す際にも Konoha スタックに引数を push する必要がある．

そこで LLVM バックエンドでは従来のインタフェースに加え，引数, 返り値を LLVM IR を用いて操作を行うインタフェース (特化インタフェース) を用意する．

以下に特化インタフェースの例として factorial の KonohaScript コードと factorial メソッド，そして特化インタフェースの例を示す．バックエンドによって生成された factorial メソッドとその特化インタフェースは読みやすさのためにバックエンド生成したコードを C 言語に書きなおしたコードを示す．

```
/* で記述された関数KonohaScriptfactorial */
int factorial(int n) {
  if (n < 2) return 1;
  else
    return n * factorial(n-1);
}
/* バックエンドで定義されたメソッドLLVM */
void fact(CTX ctx, knh_sfp_t *sfp, long _rix) {
    int n = sfp[1].ivalue;
    int ret = fact_opt(ctx, sfp, n);
    sfp[_rix].ivalue = ret;
}
/* 定義された特化インタフェース版 */
int fact_opt(CTX ctx, knh_sfp_t *sfp, int x) {
    if (n < 2)
        return 1;
    return n * fact_opt(ctx, sfp, n-1);
}
```

LLVM バックエンドで生成されたメソッドを呼び出す際には Konoha スタックの push/pop 操作を省くことができるため，このような工夫をすることで他のメソッドと同じインタフェースを保ちつつ高速に実行することが可能となる．

6. 評　　価

我々が設計，実装した LLVM バックエンドを用いて，KonohaScript アプリケーションの実行速度に与える影響について評価を行った．

6.1 実 験 環 境

本実験では以下の実験環境において，インタプリタ

と LLVM バックエンドについてプログラム実行時間の比較を行う．以下に本実験で用いた環境と，利用した C コンパイラ,LLVM ライブラリを示す．今回の測定において, KonohaJIT コンパイラは 実験環境へ移植が済んでいないため，本稿では評価は行わない．

- CPU Intel(R) Core(TM) i7 960 3.20GHz
- メモリ 12GB
- OS Ubuntu 11.04
- C コンパイラ GCC 4.4.5
- LLVM version 2.9

LLVM は部分冗長式除去や定数伝搬など様々な最適化パスを用いて LLVM IR の最適化を行うことが可能である．最適化パスの数, 適応の順番によって得られる性能は，大きく異なる．本実験では LLVM が基本セットとして用意している，関数の最適化パスのセット (StandardFunctionPass)，プログラム全体の最適化パスのセット (StandardModulePass)，をそれぞれ適応し，評価を行った．

6.2 測 定 結 果

前述した計算環境において，フィボナッチ数列の 40 番目を求めるプログラム fibo40，Shootout Benchmark[3] から nbody, mandelbrot, spectralnorm, binarytree の各プログラム, また AOBench[11] のベンチマークプログラムを KonohaScript へ移植し，適応する最適化パスを切り替えて性能評価を行った．

図 4 は各ベンチマークプログラムをインタプリタ, LLVM で実行した場合の実行時間の比較である．なお，縦軸はインタプリタとの実行時間の比を表し，それぞれ，インタプリタの実行時間，LLVM バックエンドを利用し LLVM 上で最適化を行わずコード生成を行った場合 (LLVMKonoha)，そして LLVM 上で最適化を行った場合の実行時間 (LLVMKonoha 最適化) を表している．LLVM IR に変換し，LLVM 上で最適化を行うことでインタプリタで実行した場合と比較して spectralnorm では最大で 7 倍の高速化を達成できていることが確認でき，LLVM バックエンドを用いることで性能の向上を得ることができた．

binarytree ベンチマークでは他のベンチマークと比較してあまり速度向上していない．binarytree は深さ 4〜20 の 2 分木の構築と破棄を繰り返すプログラムとなっており，オブジェクトの作成とメモリが足りなくなることで起こる GC が主なオーバヘッドである．つまり VM 以外の部分が処理時間の大部分を占めており，LLVM の最適化パスによる速度向上の効果が少ないためと考えられる．

次に，最適化パスによる最適化の効果，最適化にかかる時間の評価を行うために我々は aobench プログラムを用いて適応される最適化パスの数，最適化パスの実行時間，最適化した際の実行時間を**表 2** に示した．適応される最適化パスの数は，関数の最適化を行う場合には 5 パス，プログラム全体の最適化を行う場合には 42 パス適応される．

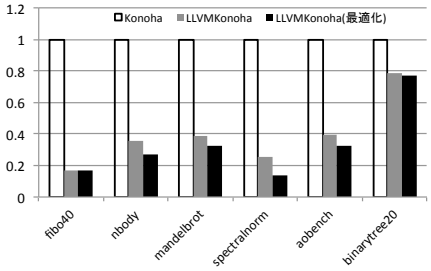

図 4 ベンチマーク結果

最適化の種類	プログラム実行時間 (sec)	コンパイル時間 (sec)	最適化パス数 (関数ごとの最適化パス/プログラム全体の最適化パス)
最適化なし	6.03	0.05	0/0
関数	5.10	0.09	5/0
関数+プログラム全体	4.84	0.19	5/42

表 2 aobench のコンパイル時間とプログラム時間の比較

関数のコンパイル時間は最大で 20msec 以下であり，実行時間の 5%以下であったため，プログラム実行時間への影響は少ないと考えられる．

7. 関 連 研 究

本節では，LLVM をコード生成器として利用しているプログラム言語と，スクリプティング言語について概観し，本研究との関連性を述べる．

近年，ahead-of-time コンパイラや just-in-time コンパイラなどコード生成器やコードの静的解析ツールとして LLVM を用いた研究が多く行われている．Terei らは Haskell 言語の実装の 1 つである Glasgow Haskell Compiler に対して LLVM を用いたバックエンドを構築している．本研究では静的型付けスクリプト言語である KonohaScript 向けのバックエンドを構築した．KonohaScript はプログラム実行時中にコード生成を行うが，GHC の LLVM バックエンドではプログラム実行前にコード生成を行う点で異なっている．

LLVM を Ruby の JIT コンパイラとして利用しているプロジェクトとして Ruby で実装された Ruby 実

行環境である Rubinius[4] がある．Rubinius ではコアライブラリ以外の機能を Ruby で記述し，プログラム実行中に LLVM を用いて機械語に変換している．また，LLVM を Python から利用するプロジェクトとして UnladenSwallow[1] がある．本研究では既存のプログラムや C 言語で実装された拡張ライブラリをを変更せずに利用できる点で両者共に異なっている．

8. ま と め

本稿では，KonohaScript の LLVM バックエンドの設計，実装について述べた．KonohaScript のプログラム，データ構造と LLVM の扱う表現，データ構造の違いについて比較し，コード変換器について説明した．マイクロベンチマーク，中規模なベンチマークプログラムから性能を評価を行いインタプリタに比べ高速な動作ができることを確認した．

今後の方針としては，SIMD などベクトル大規模アプリケーションについて LLVM バックエンドの評価を行い，短い最適化時間で効果的な最適化を施すために LLVM が適応する最適化パスの選別を行う予定である．

謝　　辞

本研究は，JST/CREST「実用化を目指した組込みシステム用ディペンタブル・オペレーティングシステム」領域の研究課題「実行時の安全性を確保する SecurityWeaver と P-SCRIPT」の一部として行われた．

参 考 文 献

1) A faster implementation of python. `http://code.google.com/p/unladen-swallow/`, 2010.
2) Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.
3) B.Fulgham. The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`.
4) Evan Phoenix. `http://rubini.us/`.
5) Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.
6) Google. V8 JavaScript Engine. `http://code.google.com/p/v8/`.
7) Masahiro Ide and Kimio Kuramitsu. コード評価器の実行時最適化を行う just-in-time コンパイラの設計と実装. 情報処理学会・プログラミング研究会. Information Processing Society of Japan (IPSJ).
8) Kimio Kuramitsu. Konohascript: A static scripting language. `http://konohascript.org`.
9) Kimio Kuramitsu. Konoha - implementing a static scripting language with dynamic behaviors. In *Workshop on Self-sustaining Systems 2010*, S3, The University of Tokyo, Japan, September 2010.
10) Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
11) Syoyo Fujita. Ambient Occlusion Benchmark. `http://code.google.com/p/aobench/`.

質疑応答

Q 今回の手法で得られた結果 (実行時間) のうちどれくらいがコンパイル時間なのか? また，コンパイル時間が短いと感じたが，いろいろな最適化のうちなにを適用したのか，その手法を教えて欲しい．

A 多くの最適化パスを関数毎に適応している．(なお，本稿で行った実験のうち，関数のコンパイル時間/最適化時間は最大で 20msec 以下であり，実行時間の 5%以下であったため，実行時間への影響は少ないと考えられる．)

Q LLVM IR に頼りすぎると，version が上がったときに大変だ，互換性はどうなっているのか?

A 数年前までは大きな変更が多く変わったりしていたが，最近は安定しているため，比較的メンテナンスコストは低く抑えられると考えている．

Q クロージャをサポートするのは LLVM IR 上で実装するよりもインタプリタ上で実装する方が容易ではないのか？

A ランタイムライブラリにクロージャをサポートする機能を追加する必要があると考えている．

Q メソッドから LLVM IR に変換すると大域的な変換ができないため，プログラム全体を見渡した上でコードの解析を行った上で，AST から LLVM に変換したほうが，プログラム全体の最適化できるのでは？

A メソッド単位で LLVM IR に変換し，最適化を行う場合には十分な最適化を行うことは難しい．(KonohaScript プログラムをすべて LLVM IR に変換し，プログラム実行前にプログラム全体の最適化を行うことで性能の向上を得ることができた．)