OpenJIT フロントエンドシステムの設計

小川宏高[†] 松岡聡[†] 丸山冬彦[†]
早田恭彦[†] 志村浩也[††]

Open Compilerは，自己反映計算をベースとして，コンパイラにさまざまな言語拡張や最適化のためのモジュールを組み込む技術である．我々は，Java 言語の Just-In-Time コンパイラに Open Compiler の技術を適用し，クラス単位での最適化のためのカスタマイゼーションを可能にした OpenJIT を開発している．OpenJIT は，アプリケーションや計算環境に特化した動的な言語機能の拡張や最適化が可能であり，新たなコンパイル技術の研究基盤としての役目を果たす．OpenJIT は，可搬性を確保するために 99% が Java 言語で記述されており，フロントエンドシステムとバックエンドシステムから構成される．前者はより高レベルな中間表現での最適化・特化を支援するバイトコード変換器のフレームワークを提供し，後者はコード生成レベルでの最適化を行い，実行時コードを生成する．本稿では，フロントエンドシステムの実現について述べるとともに，それを用いた単純な例による評価を行う．

A Design of OpenJIT Frontend System

Hirotaka Ogawa,[†] Satoshi Matsuoka,[†] Fuyuhiko Maruyama,[†]
Yukihiko Sohda[†] and Kouya Shimura [††]

The so-called 'Open Compilers' is a technique to incorporate various self-descriptive modules for language customization and optimization based on computational reflection. We apply the open compiler technique to a Java Just-In-Time compiler to develop the OpenJIT compiler, which allows class-specific customization and optimization, fostering research of new compilation techniques such as application-specific customization and dynamic optimizations. The OpenJIT is largely devided into the frontend and the backend. The frontend takes the Java bytecodes as input, performs higher-level optimizations involving source-to-source transformations, and pass on the intermediate code to the backend. The backend takes the intermediate code from the frontend as input, performs lower-level optimizations, and outputs the native code for direct execution. In this paper, we describe the internal architecture of the frontend system and evaluate it for a simple loop example.

1. はじめに

Java 言語に代表される可搬性の高いプログラム言語が近年注目されている．これらの言語の処理系では，可搬性を確保するためにバイトコードなどのコンパクトかつ機種独立な中間形式を用い，高速化のために必要な部分を実行時にネイティブコードにコンパイルし，実行速度を向上させる Just-In-Time（JIT）コンパイラを用いるのが一般的である．しかし，JIT コンパイラは，通常のコンパイラと比較すると，その構築に関する統合的な技術フレームワークが与えられていない．そのため，JIT 自身の可搬性の欠如，最適化に関する技術的知見の不足などの問題が指摘されている．

通常のコンパイラに要求される「最適化」は，メモリなどの資源が充分に確保できるという仮定の基に，なるべく高性能な実行時コードを得ることに主眼が置かれている．一方，JIT コンパイラに要求される「最適化」は，一つのプログラムを様々な異なる計算環境へ適合させるため，（1）個々のアプリケーション及び計算環境に特化した最適化，（2）計算環境とアプリケーションに応じたコンパイルコードの拡張，を考慮する必要がある．しかし，従来の JIT は一種のブラックボックスであり，（1）に関しては汎用性のある最適化しか行わず，また，（2）に関しては言語の拡張や新規の機能に対応してコード生成の手法を変えるような特化の機能は提供していない．

この問題の原因は，JIT の構築が単純に従来のコンパイラ技術を下地として，アドホックに構築されている

† 東京工業大学
Tokyo Institute of Technology
†† （株）富士通研究所
Fujitsu Laboratories Limited

ためと考えられる．Java の JIT の多くは C 言語やアセンブラで記述されており，再利用性，可搬性，適合性など，本来ソフトウェアが持つべき性質を兼ね備えていない．通常のコンパイラでは，Stanford 大学の SUIF Compiler System[13] のように，高品質なコンパイラをオブジェクト指向フレームワークとして実現し，様々な種類のコンパイラ構築の研究や技術開発の礎とする試みがあるが，通常の JIT は完全なブラックボックスであり，ソースコードすら公開されていない．このような現状が，JIT を含む動的コンパイル技術の研究の大いなる妨げになっていると我々は考える．

我々は上記の問題を解決するために，自己反映計算（リフレクション）に基づいた Open Compiler（開放型コンパイラ）[7] 技術をベースとして，アプリケーションや計算環境に特化した言語の機能拡張と最適化が行える JIT コンパイラである OpenJIT を研究・開発している．OpenJIT では，JIT コンパイラ自身が Java のクラスフレームワークとして記述されており，クラスライブラリの作成者がクラス単位でそのクラスに固有の最適化モジュールを組み込むことを可能とする．これにより，様々な計算環境・プラットフォーム・アプリケーションに対する適合や，複雑な最適化を組み込むことも可能となる．

本稿では，OpenJIT の概要について述べるとともに，OpenJIT が提供するクラスファイル単位での最適化・特化の機能を実現するフロントエンドシステムの実装について述べる．さらに，単純なループのプログラム変換を例にとって，現状の OpenJIT の予備的評価を行う．

2. OpenJIT の概要

我々が研究開発を進めている OpenJIT コンパイラシステム[9] は，仮想機械によるプログラムの実行の高速化に用いられる JIT コンパイラに自己反映計算の概念を導入した開放型 JIT コンパイラである．OpenJIT は，計算環境・プラットフォーム・アプリケーションに応じた動的なコンパイルの適合化が可能にするため，様々な計算環境に対して「性能の可搬性」や「保守性」の高いプログラムが実現できる．

これに対し，通常の Java のシステムは可搬性があると言われているが，その可搬性は JVM（Java Virtual Machine）が実現する仕様の可搬性に依存している．JVM が実現していない機能は，Java で簡便に記述できない限り，非標準的な手段によって実現せざるを得ない．

例えば，Java のマルチスレッドを用いた並列プログラムを分散メモリ計算機で実行しようとした場合，何らかの形で分散共有メモリを Java 上で実現する必要がある．ところが，JVM はそもそも分散共有メモリを一切サポートしておらず，またユーザーレベルでの分散共有メモリの実現に必要な Read/Write Barrier を挿入するためのソフトウェア上の「フック」も存在しない．従って，JVM を改造するか，プリプロセッサを用いるなど，煩雑で可搬性を妨げる手法を用いる必要がある．

OpenJIT では，コンパイラの拡張クラスファイルとして，Read/Write Barrier をコンパイルコードで一般的に実現することにより，必要な場合に分散共有メモリのコードを出力するような可搬性の高いシステムの構築が可能となる．つまり，OpenJIT は Java に JVM が提供しない一種のフックを入れる手段として利用できる．

また，OpenJIT は，より効果的な最適化をクラスに特化した形で実現することを可能にする．これにより，効果的だがコストの高い最適化を，プログラムの必要な部分だけに適用できる．

例えば，Java ではあまり行われてこなかった数値計算の最適化（ループ変換・各種キャッシュブロッキング・データ分散のアドレス計算の最適化など）を，必要なクラスの集合に対して行うことも可能となる．このような最適化は，従来の Fortran や C のコンパイラでは必須だが，JIT コンパイラにおいては全てのプログラムに対して行うにはコストが高すぎるという理由から忌避されてきた．一方，OpenJIT では，そのような最適化を，必要に応じてクラスライブラリ・アプリケーションの作成者，およびユーザが必要なクラスに対して場合に応じて行える．

さらに，アプリケーションに特化されすぎるが，しかし極めて有効な最適化が，汎用の JIT コンパイラに組み込むことができない場合にも対応が可能になる．例えば，低レベルの画像マルチメディア処理では，複数の画像処理オペレータを合成すると数十倍から数百倍の速度向上が得られることが報告されているが，ソースレベルでオペレータ合成を行うと，ソースプログラムが極めて煩雑なものとなる．そこで，何らかの特化したコンパイラが望まれるが，そのような特化したコンパイラの作成は難しく，かつソースの機密性も問題となる．OpenJIT では，バイトコードレベルでオペレータ合成を行う特殊なコンパイルをするモジュールを組み込むことによって，対応が可能となる．

このように，OpenJIT は従来のコンパイラと比較し，新たなプログラミングスタイルを可能にする．従来の JIT を前提にしたプログラミングでは，**図 1** の上図

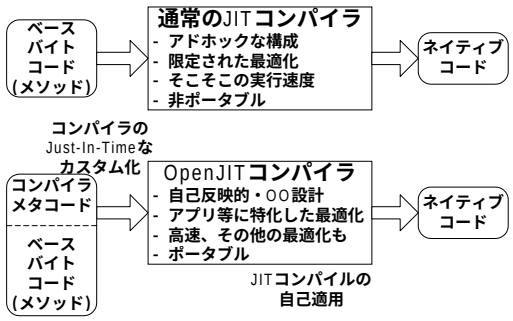


図 1 従来の JIT と OpenJIT の比較の概念図
Fig. 1 Comparison of Traditional JITs and OpenJIT

のように，ベースレベルのプログラムを JIT コンパイラがコンパイルし，プログラマやユーザは JIT コンパイラが必要な最適化をすることを期待するか，あるいは何らかのソースレベルでの変換をし，オリジナルのプログラムと比較すると見通しの悪いプログラムを保守せざるを得なかった．しかし， OpenJIT により，図 1 の下図のように，プログラマは見通しの良い Java のベースレベルのコードを記述し，プログラマ自身，あるいはそのプログラムのユーザが適切な最適化・適応化クラスライブラリを選択することにより，ベースレベルの記述の簡潔さを保存したまま，効率化を図ることができる．このような性質は自己反映システム一般に備わるものだが，JIT コンパイラとして実現することにより，従来の自己反映システムと比較して，プログラムのソースをユーザに開示することなく，ポータブルに，かつ高い実行効率を得ることが可能となる．

3. OpenJIT の構成

OpenJIT は， Java 言語自身でポータブルに記述され， Java Virtual Machine の JIT に対する標準 API を満たすように作成される．

OpenJIT の機能は，大きく分けてフロントエンドシステムの処理とバックエンドシステムの処理の 2 つから構成される．

フロントエンドシステムは，バイトコードレベル，もしくはより高レベルな中間表現での最適化器・特化器のフレームワークを提供する部分であり， Java のバイトコードを入力として受け取ると，任意の最適化，言語拡張や新規の機能に応じた特化などの高レベルな変換を施して，バックエンドシステムに渡す中間表現（あるいはバイトコード）を出力する．

一方，バックエンドシステムは， Java 言語自身でポータブルに記述された低レベルの JIT コンパイラ機能を提供する部分であり，既存の JIT コンパイラと同様，得られた中間表現（あるいはバイトコード）に対し，より低レベルでの最適化を行い，ネイティブコードを出力する．

フロントエンドシステムとバックエンドシステム間のインターフェースに中間表現とバイトコードの両者を用意したのは，システム全体の処理性能を重視するとともにモジュラリティを確保するためである．

以下では，フロントエンドシステム，バックエンドシステムの処理について概略を述べる．

3.1 フロントエンドシステム

OpenJIT の起動は， JDK の提供する JIT インターフェースを介してメソッド起動ごとに行われる． OpenJIT が与えられたメソッドに対して起動されると，フロントエンドシステムは対象となるバイトコード列に対して以下の処理を行う．

まず，対象となるバイトコード列から元の Java 言語による記述とほぼ等価な抽象構文木にデコンパイルする．この際には，与えられたバイトコード列から，元のソースプログラムから生成されるコントロールグラフの復元を行う．これと同時に，対象となるクラスファイルに何らかのアノテーション情報が付記されていた場合にその情報を得る．このアノテーション情報は必要に応じて抽象構文木上の付加情報として用いられる☆．

次に，得られた抽象構文木に対して最適化・特化などの変換を施す．これらの処理は，抽象構文木から必要なデータフローグラフを構築し，その上での解析を行い，必要に応じて変換を行うことで実現できる．フロントエンドシステムでは，（1）対応するデータ依存グラフやコントロール依存グラフを生成する機能，（2）これらのデータフローグラフに対するデータフロー問題，マージ，不動点検出などを提供する手続き群，（3）テンプレートマッチングによる抽象構文木上の変換メソッド，を提供しており，これらを用いてユーザ定義による変換器を構成することができる．

変換後の結果はバックエンドシステムが必要とする中間表現ないしバイトコードに変換されて，出力される．

3.2 バックエンドシステム

バックエンドシステムはフロントエンドシステムによって出力された中間表現ないしバイトコードに対して，低レベルの最適化処理を行い，ネイティブコードを出力する．

まず，バイトコードの命令を解析して分類することに

☆ 現状の実装では，実行時のフロントエンドの処理を抑えるために，アノテーション情報が付記されていない場合には，フロントエンドの処理を省略してバックエンドに渡され，抽象構文木の生成は行われない．

より，スタックオペランドを使った中間言語の命令列へと変換を行う．フロントエンドシステムが中間表現を出力する場合はこの形式で出力するため，この変換処理は省略される．次に得られた命令列に対し，スタックオペランドを使った中間言語から仮想的なレジスタを使った中間言語（RTL）へ変換する．この際，バイトコードの制御の流れを解析し，命令列を基本ブロックに分割する．また，バイトコードの各命令の実行時のスタックの深さを計算することで，スタックオペランドから無限個数あると仮定した仮想的なレジスタオペランドに変換する．

次に， Peephole 最適化モジュールによって， RTL の命令列の中から冗長な命令を取り除く最適化を行い，最適化された RTL が出力される．最後に SPARC プロセッサまたは， Intel x86 プロセッサのネイティブコードが出力される．出力されたネイティブコードは，JavaVM によって呼び出され実行される．

4. OpenJIT フロントエンドシステムの実現

3.1節で述べた通り， OpenJIT フロントエンドシステムは，バイトコードレベル，もしくはより高レベルな中間表現での最適化器・特化器のフレームワークを提供する．このフレームワークは，（1）デコンパイラによってバイトコードから抽象構文木を得，（2）最適化・特化に相当する抽象構文木上の変換を実施し，（3）最後に抽象構文木からバックエンドシステムの中間形式またはバイトコードを生成する，という一連の処理を実現する Java のクラスライブラリから構成される．

抽象構文木上の変換は，ツリーをトラバースしながら，必要に応じてツリーを更新するメソッドによって実現され，このメソッドが一種のフック機構として機能する． OpenJIT フロントエンドシステムでは，このフック機構をユーザの必要に応じて利用するために，クラスファイルにアノテーション情報を付加して拡張しており，この情報を「クラスファイルアノテーション」と呼んでいる．

OpenJIT フロントエンドシステムの概観は**図 2** に示す通りであり，以下の 4 つのモジュールから構成される．

（1） OpenJIT Bytecode Decompiler
バイトコードを抽象構文木に変換する機能を提供する．

（2） OpenJIT Class Annotation Analyzer
クラスファイルアノテーションを抽出し，アノテーション情報を抽象構文木に付加する機能を提供する．

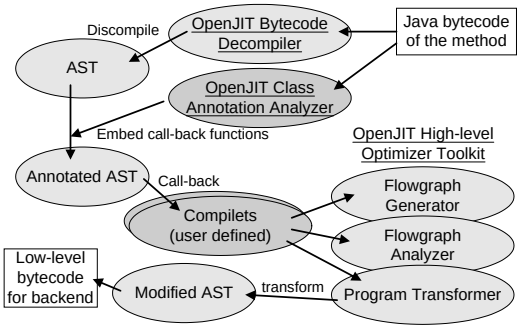


図 2 OpenJIT フロントエンドシステムの概要
Fig. 2 Overview of OpenJIT Frontend System

（3） OpenJIT High-level Optimizer Toolkit
OpenJIT を用いた最適化，コンパイルコードの特化を行うモジュール（以下では Compilets と呼ぶ）を支援するツールキットを提供する．

（4） Abstract Syntax Tree Package
抽象構文木の表現，およびユーティリティルーチンを提供する．

以下では， OpenJIT に特有の拡張であるクラスファイルアノテーションについて概要を説明した後，上記の 4 つのモジュールについて説明する．

4.1 クラスファイルアノテーション

クラスファイルアノテーションとは，最適化や特化を行うために， OpenJIT に与えるさまざまな付加情報のことである．

クラスファイルアノテーションに含まれる付加情報には，例えば以下のものがあり得る:

ユーザ定義による最適化器・特化器クラスのサポート
OpenJIT では，フロントエンドシステムの振る舞いを変更する最適化器・特化器クラスをユーザが定義することができる．クラスファイルアノテーションでは，この最適化器・特化器クラスの指定を与えることができる．具体的には，クラスファイルの名前を与える，クラスファイルアノテーションに最適化器クラス自体をエンコードするなどの方法によって実現可能である．

分散共有メモリのサポート OpenJIT を用いてユーザーレベルでの分散共有メモリを実現する場合， Read/Write Barrier を適切に挿入する必要がある．しかし，静的な解析によって immutable と分かっているオブジェクトに対しては，実行時にこれらの処理を省略することができる．このような場合， immutable なオブジェクトに関する一覧表をクラスファイルアノテーションに含めることができ，不要な実行時処理を避けることができる．

数値計算の最適化のサポート Java の配列を用いた行列計算などでは，Array Bounds Check の必要性などから，Fortran などのコンパイラでは行われている多重ループの自動的な変換が難しい．安全に変換でき，かつ効果的であることが静的に判定できるループにマーキングを行うことで，実行時にループ変換・各種キャッシュブロッキング・データ分散のアドレス計算の最適化などを行うことができる．

クラスファイルアノテーションの実現には，クラスファイルに含まれる各メソッドのアトリビュート領域に付加するという方法を採る．JVM の規約により，JVM が認識しないアトリビュートは無視するため，この方法で作成されたクラスファイルアノテーション付きクラスファイルは，OpenJIT 以外の JIT コンパイラでも支障なく実行でき，クラスファイル自体の可搬性も維持される．

クラスファイルアノテーションは，各メソッドのアトリビュート領域に付加されるが，クラスファイルアノテーションの性質上，クラスファイルに共通する情報に関してはクラスのアトリビュート領域に保持することが望ましい場合がある．これは，JDK の標準の JIT インターフェースが各メソッドのアトリビュート領域へのアクセスしか許していないことに起因する制限であり，JVM 自体の変更なしには実現が難しい．

また，OpenJIT が入力として仮定するクラスファイルには，ネットワークなど通じて一般的に流布している標準的なもの（標準クラスファイル）と，拡張が施されたクラスファイルアノテーションを含むもの（拡張クラスファイル）とに区別できる．フロントエンドシステムによるプログラム変換は，より多くの，様々な形態で流布されるクラスファイルに適応可能であるべきであり，後者に限定されるべきではない．

ソースを含む形態で配布されるプログラムに対しては，拡張クラスファイルを生成するために，その機能を有する特殊な Java 言語コンパイラ（javac）を用いることで可能になる．

一方，ソースを含まない形態で配布される標準クラスファイルに対しては，簡便かつ自動化された手続きでクラスファイルアノテーションを付加するクラスファイル変換ツールを用いるか，あるいは OpenJIT の設定として，特定のクラスファイルに特定のプログラム変換を行う指示を与えることによって，対応し得る．

この両者を比較すると，前者はより詳細な情報や指示をアノテーションとして与え得るため，フロントエンドシステムでの処理の容易性や多様性に寄与する．一方，後者は限定的な情報や一律な指示しか与え得ないため，フロントエンドシステムでの処理を限定する．特に上記のアノテーションの利用例では，分散メモリや数値計算の最適化のサポートは限定的にならざるを得ない．

OpenJIT の設計方針としては，ユーザが行いたいプログラム変換の性質やソースプログラムの利用可能性に柔軟に適応できるように，この 2 つの方法の一方に利用法を限定しない．

4.2 OpenJIT Bytecode Decompiler モジュール

OpenJIT Bytecode Decompiler モジュールは，バイトコードを入力とし，抽象構文木に変換して出力する機能を提供する．このモジュールでは，以下の手順で処理を行う．

(1) メソッドのバイトコードを JVM の命令列に分解し，同時にベーシックブロックのリーダとなるべき命令に印をつける．
(2) ベーシックブロックのリストとしてコントロールフローグラフを構築する．
(3) コントロールフローグラフに対応するドミネータツリーを構築する．
(4) ベーシックブロック毎のシンボリック実行により，ベーシックブロックをまたがない（部分）式や文を復元する．
(5) 複数のベーシックブロックと条件分岐により実現される，Java 言語の`&&` や`||` のような演算子による式や条件式を表すコントロールフローを見つけ出し，それらの式を復元する．
(6) 制御構造を復元する．
(7) 結果を抽象構文木として出力する．

これらの内，(1) から (5) および (7) は単純であるか，既知の手法を用いて実現できる．我々は，効率的に制御構造を復元する新しいアルゴリズムとして，直接コントロールフローグラフを解析する代わりに，対応するドミネータツリーを用いる方式を採用して，効率的に制御構造の復元を実現している[14]．この方法は，従来の，Krakatoa[11] に代表されるパターンマッチングを用いて制御構造を復元する手法と比較して，効率が良く，code obfuscation に対してロバストな性質を持つ．

4.3 OpenJIT Class Annotation Analyzer モジュール

OpenJIT Class Annotation Analyzer モジュールは，クラスファイルからクラスファイルアノテーションを抽出し，アノテーション情報を抽象構文木に付加する機能を提供する．このモジュールでは，以下の手順で処理を行う．

(1) メソッドのアトリビュート領域にアクセスして，

アトリビュートのバイトアレイを取得する．これは，Native Method として実現されており，JDK 1.1.x の VM から得られるメソッドブロックの領域をパーズすることによって得られる．

(2) このバイトアレイを，Annotation オブジェクトがシリアライズされているものとして readObject メソッドで読み込むことで，Annotation オブジェクトを生成する．

(3) Annotation オブジェクトを，抽象構文木のノードにアノテーション情報として登録する．

クラスファイルアノテーションの用途によって，含まれる情報の仕様が異なるため，サブクラス化によって汎用性・拡張性を確保する．スーパークラスの Annotation クラスには，アノテーションの種類を識別する文字列と登録先になる抽象構文木のノードを共通に保持する．

4.4 OpenJIT High-level Optimizer Toolkit

OpenJIT High-level Optimizer Toolkit は，OpenJIT を用いた最適化，コンパイルコードの特化を行うモジュール（Compilets）の構築を支援するツールキットである☆．このツールキットは，抽象構文木とフローグラフを用いる一般的なコンパイラの最適化技法に使われる機能を実現した，以下の 3 つのモジュールから構成される．

(1) Flowgraph Constructor
Flowgraph Constructor は，抽象構文木からデータ依存グラフ，コントロール依存グラフなどを含む Flowgraph オブジェクトを生成する機能を提供する．フローグラフの生成器は，Factory Method パターンに従い，抽象構文木のルートノードを引数に取って，Flowgraph クラスのサブクラスのコンストラクタとして実現される．従って，他の種類のフローグラフの生成器もユーザによる拡張によって容易に実現できる．

(2) Flowgraph Analyzer
Flowgraph Analyzer は，フローグラフ上の一般的な計算を行うフローグラフの解析アルゴリズムルーチン（データフロー問題，マージ，不動点検出など）を提供する．これらのルーチンは，Command パターンに従い，各アルゴリズムごとに Analyzer クラスのサブクラスに分割されており，各サブクラスの Execute メソッドとして，実装される．従って，OpenJIT で提供されていないアルゴリズムもユーザによる拡張によって容易に追加できる．

(3) Program Transformer
Program Transformer は，パターンマッチによる抽象構文木から抽象構文木への変換を行う．この変換器は，以下の手続きからなる．

- register_pattern(Expression src, Expression dst)
- register_pattern(Statement src, Statement dst)
  変換前・変換後の抽象構文木のパターンを指定して，パターンマッチのルールを登録する．
  register_pattern メソッドは，抽象構文木の部分式または文を表す，変換前・変換後のパターンを登録する．これらのパターンは Abstract Syntax Tree Package を用いて生成することができる．
- substitution(Expression root)
- substitution(Statement root)
  指定されたノードをルートノードとする抽象構文木のサブツリーに対して，深さ優先探索で登録されたパターンとマッチする部分を探索し，マッチすれば，置き換えを行う．

このパターンマッチによる変換は，低レベルであり，パターンを生成する手続きも煩雑である．柔軟性を確保するためには，より高度な記述からパターンを生成する API を用意することなどが考えられる．

4.5 Abstract Syntax Tree Package

Abstract Syntax Tree Package は，抽象構文木の表現，およびユーティリティルーチンを提供する．フロントエンドシステムで用いる抽象構文木は，Java 言語とほぼ等価であり，抽象構文木の各ノードは Java 言語の抽象構文の 1 つ式ないし文を表す．このパッケージでは，Java 言語のほぼすべての構成子に対応する特化されたクラス（100 クラス以上）が適切に階層的サブクラス化されて用意されている（**図 3**）．

図 4，**図 5** にそれぞれ，典型的な Expression と Statement を表すクラスの構成要素を示す．

典型的な 2 項演算式を表す Expression クラスは，オペレータの種類を表す ID，左辺式，右辺式，式全体の型，アノテーションへの参照を保持する．アノテーションへの参照は，OpenJIT Class Annotation Analyzer によって，必要があれば適切なアノテーションへの参照に書き換えられる．code メソッドは，この式をルートノードとする部分式に対応するバックエンド用の中間形

☆ OpenJIT の初期のディストリビューションでは，Compilets 自体は含まない．

- Node
  - Expression
    - BinaryExpression
      - AddExpression
      - SubtractExpression
      - MultiplyExpression
      - ...
    - UnaryExpression
    - ConstantExpression
    - ...
  - Statement
    - IfStatement
    - ForStatement
    - WhileStatement
    - CaseStatement
    - ...

図 3 Abstract Syntax Tree Package のクラス階層
Fig. 3 Class Hierarchy of Abstract Syntax Tree Package

式またはバイトコードを生成するメソッドであり，通常の Java 言語コンパイラ（javac）のコード生成器とほぼ同じ働きをする．code メソッドは，右辺式・左辺式の code メソッドを再帰的に呼び出すことによってコード生成を行う．その過程でアノテーションへの参照が null でない場合には，この式に関連付けられたアノテーションの execute メソッドをコールバックすること，最適化・特化などの処理を挿入するフックとして機能する．

また，If 文を表す Statement クラスは，Expression とほぼ同様に，文の種類を表す ID，条件式，Then 部の文，Else 部の文，アノテーションへの参照を保持し，この If 文に対応するバックエンド用の中間形式またはバイトコードを生成する code メソッドを持つ．

Abstract Syntax Tree Package を用いて作られた抽象構文木は，OpenJIT High-level Optimizer Toolkit を用いて作成された Compilets によって，解析や変換の対象となる．この解析や変換を容易にするという要請からは，抽象構文木の表現に SSA 形式（Single Static Assignment Form）を採ることが望ましい．一般的に SSA 形式を用いると，データフロー解析や，定数伝搬，共通部分式除去，ループ最適化，コード移動などの典型的な変換を単純にすることができる．現状の 4.2 OpenJIT Bytecode Decompiler モジュールで生成される抽象構文木は通常の抽象構文木であるが，これを SSA 形式に変更することは困難ではないと考えている．

5. 予備的評価

OpenJIT フロントエンドシステムによる単純なプログラム変換例を用いて，本システムの予備的評価を行う．ただし，現状の実装ではフロントエンドシステムとバックエンドシステムの間のインターフェースにバイトコードを用いており，バックエンドシステムの中間表現を直接生成する方法に比べて，効率は低い．

```
public int[][] matmul(int[][] m1, int[][] m2) {
  for (int i = 0; i < SIZE; ++i) {
    for (int j = 0; j < SIZE; j++) {
      for (int k = 0; k < SIZE; k++) {
        T[i][j] += m1[i][k] * m2[k][j];
      }
    }
  }
  return T;
}
```

図 6 行列積メソッド（変換前）
Fig. 6 Matrix Multiply method (before)

評価のために，**図 6** に相当するプログラムを**図 7** に相当するプログラムに変換する例を考える☆．この例では，クラスファイルアノテーションとして，matmul メソッドのアトリビュート領域に「LoopTransformer」という文字列の String オブジェクトを Serialize して格納した．ここで，このアノテーションをフロントエンドシステム側では，メソッド本体のルートノードに対して LoopTransformer を適用するという意味で用いることにする．この拡張クラスファイルの生成には，任意のオブジェクトを指定したメソッドのアトリビュート領域に格納するツールを作成して利用した．一方，特化器は，Annotation クラスのサブクラス LoopTransformer として作成する．LoopTransformer クラスの execute メソッドで，完全ネストした 3 重ループの最内ループにある 2 次元配列で，第 1 次元だけが最外ループのループ変数に束縛されるものを見つけて，置換を行う．LoopTransformer クラスの概略を**図 8** に示す．完全な記述には 200 行程度要しており，特化器の定義は現状では容易とは言い難いが，共通ルーチンをクラスライブラリ化することで省力化が期待できる．

5.1 評価環境

評価環境には，以下の実行環境を用い，JDK 1.2.2（ClassicVM）に付属の JIT コンパイラ（sunwjit）との比較を行った．OpenJIT では，図 6 のプログラムをそのまま実行させた場合と，実行時にフロントエンドシステムによって図 7 相当のプログラムに変換して実行させた場合を測定した．sunwjit での測定では，変換前と変換後のソースプログラムを javac で予めコンパ

☆ 読み易さのためにソース形式で示した．実際には，「相当する」抽象構文木上の変換を行う．

```
public class MultiplyExpression extends BinaryExpression {
   int op;                          // Construct ID
   Expression left;                 // LHS expression
   Expression right;                // RHS expression
   Type type;                       // Type of this expression
   Annotation ann;                  // Embedded Annotation (default: null)

   void code() {                    // Convert AST to backend-IR form
                                    // (or bytecodes)
      if (ann) ann.execute(this); // call-back for metacomputation
      left.code();                  // generate code for LHS
      right.code();                 // generate code for RHS
      add(op);                      // generate code for "operator"
   }

   Expression simplify() {}         // Simplify expression form
                                    // (e.g. convert "a * 1" to "a")
   ...
}
```

図 4 典型的な二項演算式に対応する Expression クラスの例
Fig. 4 An example of Expression class for typical binary expression

```
public class IfStatement extends Statement {
   int op;                          // Construct ID
   Expression cond;                 // Condition expression
   Statement ifTrue;                // Statement of Then-part
   Statement ifFalse;               // Statement of Else-part
   Annotation ann;                  // Embedded Annotation (default: null)

   void code() {                    // Convert AST to backend-IR form
                                    // (or bytecodes)
      l1 = new Label();
      if (ann) ann.execute(this); // call-back for metacomputation
      codeBranch(cond, l1);         // generate code for Condition
      ifTrue.code();                // generate code for Then-part
      l2 = new Label();
      addGoto(l2);                  // escape from Else-part
      addLabel(l1);
      ifFalse.code();               // generate code for Else-part
      addLabel(l2);                 // add label for "Break" statement
   }

   Statement simplify() {}          // Simplify statement form
                                    // (e.g. if (true) S1 S2 => S1)
   ...
}
```

図 5 If 文に対応する Statement クラスの例
Fig. 5 An example of Statement class for “If” statement

イルしておいて実行させた．行列のサイズ（SIZE）は 200×200 および 600×600 とした．

- Sun Ultra60（UltraSparc II 300MHz×2, 256MB）
- 日本語 Solaris 2.6
- JDK 1.1.8 と付属の Java 仮想マシン（ClassicVM）

5.2 評価結果

表 1 に OpenJIT と sunwjit をそれぞれ用いた場合の，matmul メソッド部分の変換前・変換後の実行時間と，matmul メソッド部分の実行開始までに要するセットアップ時間を示す．sunwjit の場合は 2 つのバージョンを予めコンパイルしたものを用いているので，セットアップ時間に差はない．

変換前・変換後のバージョンの OpenJIT と sunwjit の matmul メソッド本体の実行時間の差は 10% 以下であり，大きくはない．この差は生成されるネイティブ

```
public class LoopTransformer extends Annotation {
  int loop_nest = 0;
  LocalField index;
  LoopTransformer() {}
  boolean isRegularForm(Statement init, Expression cond, Expression inc) {
    // For 文の初期化文，条件式などを検査し，ループが正規形をしているかどうか検査
  }
  void execute(Node root) {
    if (root instanceof CompoundStatement) {
      for (int i = 0; i < root.args.length; i++) { execute(root.args[i]); }
    }
    // 完全ネストの 3 重ループか否か検査
    else if (root instanceof ForStatement &&
             root.body instanceof ForStatement &&
             root.body.body instanceof ForStatement) {
      if (isRegularForm(root.init, root.cond, root.inc) &&
          isRegularForm(root.body.init, root.body.cond, root.body.inc) &&
          isRegularForm(root.body.body.init, root.body.body.cond, root.body.body.inc)) {
          // root のループ変数を index に記録
          // root.body.body が ForStatement を含まないことを検査
          // 含まなければ，右辺式を順に走査して，（[] （[] index） _）という形状の 2 次元配列を探索
          // 発見できれば，所定の変換を実施

    } } }
    else return;
} }
```

図 8 変換クラスの定義の概略
Fig. 8 Overview of LoopTransformer

```
public int[][] matmul(int[][] m1, int[][] m2) {
  for (int i = 0; i < SIZE; ++i) {
    int tmp[] = m1[i];
    for (int j = 0; j < SIZE; j++) {
      for (int k = 0; k < SIZE; k++) {
        T[i][j] += tmp[k] * m2[k][j];
      }
    }
  }
  return T;
}
```

図 7 行列積メソッド（変換後）
Fig. 7 Matrix Multiply method (after)

コードの効率の差であり，より適切なコード生成を行うことで改善が期待される．

変換前の OpenJIT のセットアップ時間は 1.09 秒であり，sunwjit の 0.49 秒と比較して 2 倍以上要している．OpenJIT では対象クラスファイルの他に，OpenJIT 自体もコンパイルする必要があることを考慮すれば，許容し得るオーバーヘッドと言える．

変換前・変換後の OpenJIT のセットアップ時間の差の 1.59 秒は，フロントエンドシステムでの処理に要する時間である．行列サイズが 200 の場合には，セットアップ時間の増加が速度向上を上回るのに対し，600 の場合には，逆に速度向上が上回る．現状のフロントエンドシステムの実装自体，デコンパイルの過程で少なからぬ数の小オブジェクトの生成・破棄を行うなど，必ずしも効率的ではなく，セットアップ時間に関して改善の余地が十分にある．また，フロントエンドシステムの処理は，クラスファイルアノテーションが付加されていないクラスファイルには適用されないために効率を低下させない．また変換後のコードが繰り返し再利用されることで，フロントエンドシステムの処理によるセットアップ時間の増加もいずれ償却されると考えられる．

表 1 評価結果 (単位: 秒)
Table 1 Results

matrix size	200		600	
	before	after	before	after
OpenJIT	2.52	2.26	85.22	77.74
OpenJIT setup-time	1.09	2.68	1.09	2.67
sunwjit	2.34	2.06	80.19	73.55
sunwjit setup-time	0.49	0.49	0.49	0.49

6. 関連研究

JIT コンパイラは，古くは Lisp や Smalltalk などの，中間言語を有する言語処理系の高速化技術として開発された（ParcPlace Smalltalk の Deutch-Schiffman の最適化技法など[4]）．しかし，これらは Java における JIT コンパイラと同様，C 言語などの低レベルな言語で記述され，自己反映的な特化の機能はない．

自己反映計算は，1982 年に Indiana 大学（当時 MIT）の Brian Smith の提唱した 3-Lisp によって最初の理論的な基盤が提唱された．基本的には，適切な計算系の定義により，通常の計算を表す従来からのベースレベルのコードに加え，自己の計算系を表すメタレベルのコードをユーザは記述することが可能となる．Smith の研究以後，様々な研究により，プログラミング言語やオペレーティングシステム，あるいは分散システムにおいて，それらの計算 / 実行モデルを自己反映計算モデルとすることにより，従来では例外的かつ固定的な形で扱われてきた各種の機能（例外処理，資源管理など）を，整合的かつ柔軟な方法でモデルに導入できることが示された．これらの言語や OS は，簡潔性とともに非常に高い拡張性を提供している．また，自己反映的なプログラミング技法の応用として，プログラム言語処理系のプロトタイピング，ウィンドウシステム記述や，組込み型の OS の構成，分散システムにおける数々の資源制御などが報告されている．

Open Compiler は，コンパイラのコンパイル時の挙動を自己反映的に変更する機能を提供するシステムである．コンパイラをオブジェクト指向設計に基づきモジュール化してユーザから変更可能にし，さらにコンパイル時にメタ計算をあらかじめ静的に行うことによって，言語の拡張やアプリケーション固有の最適化などを行う．古くは Lisp のマクロがアドホックに Open Compiler 的な機能を提供したが，自己反映計算系としての Open Compiler は Xerox PARC の Anibus/Intrigue[10] が先駆的であり，MPC++[6]，OpenC++ v.2.0[2]，EPP[5]，Javassist[3] など，様々な処理系が提案・研究されている．また，並列オブジェクト指向言語 ABCL/R3[8] において，JavaVM のようなインタプリタによる解釈実行に基くメタレベル記述を，部分計算（Partial Evaluation）によってコンパイルする技法が提案されている．また，Consel らは Java において部分計算を用いて高速化を図る手法を提案している[12]．

OpenJIT は，今までの Open Compiler とは異なり，JIT の各モジュールを実行時に置き換え，自己適用的な最適化を行うので，ABCL/R3 と Open Compiler の中間的な存在であると言える．従って，両方の技術を基盤とし，かつ融合させることによって，高速な処理系を構築することを目指しているとも言える．また，EPP，Javassist のような Open Compiler（あるいはそれに準ずるようなツール）が OpenJIT のフロントエンドとして，アノテーションを含んだクラスファイル，あるいは最適化器・特化器を生成することも技術的には可能であり，相補的な技術であると言える．

Stanford 大学の SUIF Compiler System[13] は，最適化コンパイラをオブジェクト指向フレームワークとして，C++ 言語で実現したものである．Suif は，実際に様々なコンパイル技術研究の基盤として利用されており，OpenJIT が JIT コンパイラ技術研究において目標とするものに近い．Suif は共通の中間表現に対する解析器・変換器をコンパイルパスに追加することで，最適化コンパイラの拡張ができる．しかし，どのコンパイルパスを利用するかという制御は，ユーザがコンパイル時に毎回与えるか，静的な戦略で行わざるを得ない．一方，OpenJIT では，どの最適化器・特化器を用いるかという選択は，クラスファイルアノテーションの形でアプリケーションに特化して与えることができ，より実用的であると言える．また，Suif では解析器の自動生成系などのツール群が提供されるが，一方でユーザが解析器を直接記述するのは不可能なほどに複雑である．OpenJIT では，自動生成系を用意するのではなく，むしろ Open Implementation に忠実に，最適化器などの生成を支援するデザインパターンや API を用意するという方針である．本稿で説明したフロントエンドシステムはこれらの実現を考慮し，単純かつ拡張性の高い実装になっている．

既存の JIT コンパイラでは，OpenJIT のように拡張性があり，自己改変可能なシステムはないが，コンパイル時にアノテーションを施したクラスファイルを生成して，その情報を利用する Annotation-directed な JVM や JIT コンパイラは複数提案されている．しかし，AJIT[1] に代表されるように，それらの多くは実行時コード生成の効率の向上を目的とし，バイトコード命令ごとにアノテーションを施すシステムであるため，OpenJIT のように JIT コンパイル自体のカスタマイズは不可能であり，より高いレベルの表現上のプログラム変換の実現も困難である．

7. ま と め

我々は，自己反映計算（リフレクション）に基づいた Open Compiler（開放型コンパイラ）[7] 技術をベースとして，アプリケーションや計算環境に特化した言語の機能拡張と最適化が行える JIT コンパイラである OpenJIT を研究・開発している．OpenJIT では，JIT コンパイラ自身が Java のクラスフレームワークとして記述されており，クラスライブラリの作成者がクラス単位でそのクラスに固有の最適化モジュールを組み込むことを可能とする．これにより，様々な計算環境・プラットフォーム・アプリケーションに対する適合や，複

雑な最適化を組み込むことも可能となる．

本稿では，OpenJIT の概要について述べるとともに，OpenJIT が提供するクラスファイル単位での最適化・特化の機能を実現するフロントエンドシステムの実装について述べた．さらに，単純なループのプログラム変換を例にとって，現状の OpenJIT の予備的評価を行った．評価結果では，OpenJIT システム自体を実行時にコンパイルする時間，およびフロントエンドシステムの処理に要する時間がかかる分，sunwjit に比べてセットアップ時間が増大した．しかし，前者のオーバーヘッドは比較的小さく，また，対象問題サイズが大きい場合や変換後のコードが十分に再利用される場合には，後者のオーバーヘッドは償却され得る．また，フロントエンドシステムの処理時間の長さは既知の実装上の問題に起因する部分があるので今後の改善も可能である．

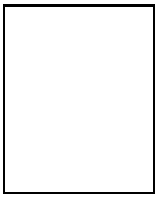
OpenJIT はまだ開発段階にあるが，既に最新の商用 JIT コンパイラ並の性能を示しており，かつコンパクトでコードも分かりやすい．本稿で述べたクラスフレームワークに加え，API を整備することによって，JIT コンパイラの研究の礎となり，かつ実用的にも様々な特殊化された適合化，最適化が可能なシステムを目指したい．今後の詳細は `http://www.openjit.org/` を参照のこと．

謝辞 本研究は，情報処理振興事業協会（IPA）の高度情報化支援ソフトウェア育成事業において，テーマ名「自己反映計算に基づく Java 言語用の開放型 Just-in-Time コンパイラ OpenJIT の研究開発」として実施された．

参 考 文 献

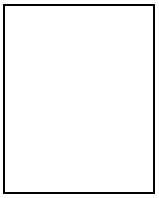
1) Azevedo, A., Nicolau, A. and Hummel, J.: Java Annotation-Aware Just-In-Time (AJIT) Compilation System, *Java Grande '99* (1999).
2) Chiba, S.: A Metaobject Protocol for C++, *Proceedings of ACM OOPSLA'95*, pp. 285–299 (1995).
3) Chiba, S.: Javassist–A Reflection-based Programming Wizard for Java, *OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (1998).
4) Deutsch, L. P. and Schiffman, A.: Efficient Implementation of the Smalltalk-80 System, *Proceedings of POPL'84* (1984).
5) Ichisugi, Y. and Roudier, Y.: The Extensible Java Preprocessor Kit and a Tiny Data Parallel Java, *Proceedings of ISCOPE'97*, Springer LNCS, No. 1343, Marina Del Rey, CA, pp. 153–160 (1997).
6) Ishikawa, Y. and et. al.: Design and Implementation of Metalevel Architecture in C++ –MPC++ Approach–, *Proceedings of Reflection'96*, San Francisco, pp. 141–154 (1996).
7) Kiczales, G., Lamping, J., Rodriguez, L. and Ruf, E.: An Architecture for an Open Compiler, *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, Tokyo, Xerox PARC, pp. 95–106 (1992).
8) Masuhara, H., Matsuoka, S., Asai, K. and Yonezawa, A.: Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages using Partial Evaluation, *Proceedings of ACM OOPSLA'95*, pp. 300–315 (1995).
9) Matsuoka, S., Ogawa, H., Shimura, K., Kimura, Y., Hotta, K. and Takagi, H.: OpenJIT–A Reflective Java JIT Compiler, *OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (1998).
10) Rodriguez Jr., L. H.: A Study on the Viability of a Production-Quality Metaobject Protocol-Based Statically Parallelizing Compiler, *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, Tokyo, p. 107112 (1992).
11) Todd Proebsting and Scott Watterson: Krakatoa: Decompilation in Java, *COOTS '97* (1997).
12) Volarschi, E., Consel, C. and Cowan, C.: Declarative Specialization of Object-Oriented Programs, *Proceedings of ACM OOPSLA '97*, pp. 286–300 (1997).
13) Wilson, R., French, R., Wilson, C., Amrasinghe, S., Anderson, J., Tjiang, S., Liao, S.-W., Tseng, C.-W., Hall, M., Lam, M. and Hennessy, J.: The SUIF Compiler System, Technical report, Computer Systems Laboratory, Stanford University (1994).
14) 丸山冬彦, 小川宏高, 松岡聡: Java バイトコードをデコンパイルするための効果的なアルゴリズム, 情報処理学会プログラミング研究会 論文誌 (掲載予定) (1999).

(平成11年 7 月16日受付)
(平成11年10月10日採録)



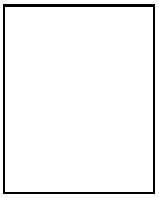
小川　宏高（正会員）

昭和 46 年生．平成 6 年東京大学工学部計数工学科卒業．平成 8 年同大学大学院工学系研究科情報工学専攻修了．平成 10 年度同博士課程中退．現在，東京工業大学大学院情報理工学研究科数理・計算科学専攻助手．プログラミング言語処理系，オブジェクト指向技術，並列計算機アーキテクチャ，広域分散システムに興味を持つ．ACM 会員．



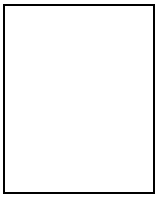
松岡　　聡（正会員）

昭和 38 年生．昭和 61 年東京大学理学部情報科学科卒業．平成元年同大学大学院博士課程中退．同大学情報科学科助手，情報工学専攻講師を経て，平成 8 年より東京工業大学情報理工学研究科数理・計算科学専攻助教授．理学博士．オブジェクト指向言語，並列システム，リフレクティブ言語，制約言語，ユーザ・インタフェースソフトウェアなどの研究に従事．現在進行中の代表的プロジェクトは，世界規模の高性能計算環境を構築する Ninf プロジェクト，計算環境に適合・最適化を目指す Java 言語の開放型 Just-In-Time コンパイラ OpenJIT，制約ベースの TRIP ユーザ・インタフェースなど．並列自己反映型オブジェクト指向言語 ABCL/R2 の研究で 1996 年度情報処理学会論文賞受賞．1997 年はオブジェクト指向の国際学会 ECOOP'97 のプログラム委員長を努める．ソフトウェア科学会，ACM，IEEE-CS 各会員．



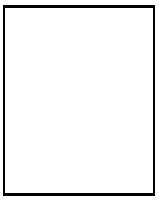
丸山　冬彦

昭和 46 年生．平成 11 年東京工業大学理学部情報科学科卒業．現在，同大学大学院情報理工学研究科数理・計算科学専攻修士課程在学中．オブジェクト指向言語，並列・分散システム，広域分散システムなどに興味を持つ．



早田　恭彦

昭和 49 年生．平成 10 年東京工業大学理学部情報科学科卒業．現在，同大学大学院情報理工学研究科数理・計算科学専攻修士課程在学中．並列・分散システム，オブジェクト指向言語，広域分散システムなどに興味を持つ．



志村　浩也

昭和 62 年京都大学工学部情報工学科卒業．平成元年同大学大学院工学系研究科情報工学専攻修士課程終了．同年，(株) 富士通研究所入社．第 5 世代プロジェクトで PIM/P のハードウェア設計，マイクロプロセッサの開発，マイクロアーキテクチャの性能評価の研究を経て，現在，Java の JIT コンパイラを研究開発に従事．