# JVMをターゲットとするコード生成のための生成系用モジュール

落合 啓二† 舞田 純一† 中井 央† 佐藤 聡†
†筑波大学

## 1 はじめに

近年、領域に特化したプログラミング言語の必要性が増し、その処理系であるコンパイラの作成の機会も増加している。しかし現状ではコンパイラを作ることは、作成の知識があまりないような人にとって簡単ではない。

コンパイラのコード生成部の作成では、比較的単純なコード生成器の場合でも、適切な対象マシンの命令群を選択し、ソース言語の機能を実現する目的コードを構築する必要がある。ここでは対象マシンに関するある程度の知識が必要となり、作成はそれほど容易でない。

そこで本研究では、ターゲットマシンに関する知識があまりないような人でも、比較的容易にコード生成部の作成ができることを目的として、コンパイラ生成系向けの中間表現モジュールを作成した。

中間表現はソース言語からの変換が容易になるよう、条件分岐、繰り返しといった手続き型言語に共通する諸機能に、オブジェクト指向言語に共通するインスタンス生成やインスタンス・メソッドの起動などを加えた、高水準な中間表現を設定した。

また本研究では中間表現の変換対象に、比較的高水準な目的言語ではあるが、可搬性に優れているJVM [1]のバイトコードを選んだ。

## 2 xMIR

我々は、中間表現の処理系の作成を支援する機構としてxMIRを開発した。xMIRは、Rubyベースの DSLであり、基本要素の定義を行うdefine_element、マクロ定義を行うdefine_macroという命令と、要素の順番を局所的、大局的に入れ替えるPROGN、INITという命令を備えている。

利用者はdefine_element命令を用いて恣意的に要素を定義できる。ここではRubyで、その要素に対する目的コードの生成やその他の処理を記述することができる。ジャンプ命令である中間表現JUMPの定義例を図1に示す。

一方define_macroでは、要素として定義した任意の命令を組み合わせた命令を定義できる。具体的には

```
define_element(:JUMP, [:label]) do
#jump に対応する目的コード
  "goto #{@label}"
end
```

図1: xMIRのdefine_elementを用いたサンプル

```
define_macro(:IF,
 [:cond, :then_body, :else_body]) do
  lab_else = tmp_name('else')
  lab_end = tmp_name('end')
  #複数の要素を組み合わせた処理を定義
  PROGN[
    JUMPC[@cond, lab_else],
    @then_body,
    JUMP[lab_end],
    DEFLABEL[lab_else],
    @else_body,
    DEFLABEL[lab_end]
  ]
end
```

図2: xMIRのdefine_macroを用いたサンプル

PROGN命令を用いて、複数の基本要素を組み合わせた処理をまとめ、その他の処理をRubyを用いて記述できる。define_macroで定義された命令も一つの要素としてみなされる。if文に相当する中間表現IFのマクロを用いた定義例を図2に示す。

## 3 モジュールの作成方法と実装

本研究の中間表現モジュールの実装には、前節で示したxMIRを用いた。

本研究で作成する中間表現は、ソース言語として手続き型および、それをベースにしたオブジェクト指向言語を想定した。そして基本セットとして、想定するソース言語が持つ諸機能にほぼ一対一で対応する高水準な表現を用意し、JVMの演算（オペランド）スタックや型の種類に応じた命令群などを変換モジュール内部で吸収することで、利用者はターゲットを極力意識せずに利用できるようにした。具体例は次章で示す。

中間表現モジュールは、前章で示したxMIRを用いて実装を行った。一般に高水準な中間表現を設定した場合、メリットとして中間表現への変換が容易である一方、デメリットとしてマシン語に近い記述性が失われてしまうということがある。そこで本研究では、目

†University of Tsukuba

的コード水準の要素をxMIRの基本要素として定義し、ソース言語水準の高水準な基本セットの要素の処理をxMIRのマクロ定義命令を用いて基本要素を組み合わせて定義した。これにより、利用者はソース言語の高水準な諸機能に対応する基本セットのほかに、必要に応じて基本要素そのものを用いることで比較的詳細な記述も可能である。

また作成したモジュールは我々が開発した自己拡張可能なコンパイラ生成系depagerから利用可能である。depagerは拡張と呼ばれるプラグインを追加することで、シンプルな構文解析器に、様々な機能を付加した構文解析器を生成可能であり、またこの拡張自体もdepagerによって作成可能である。本研究で作成したモジュールはこの拡張とともに使用可能である。depagerの詳細は参考文献[2]を参照されたい。

## 4 適用例

本研究で作成したモジュールの適用例として、pl0'[3]と、Javaのサブセットである FeatherWeightJava[4]のコンパイラの実装を行った。

JVM向けのコード生成を行う場合、pl0'のような手続き型言語のコンパイラを作成する場合でもクラス定義のコードの構築が必要となり、暗黙のObjectクラスの継承、コンストラクタのコードの構築といった煩雑な記述が必要となる。

我々の中間表現で定義したクラス定義の要素CLASSはこのような暗黙のうちに継承されるクラスの記述、コンストラクタのコードの構築を補完する機能を備えている。手続き型言語のコード生成部を作成する場合には、CLASS要素を用いると、以下のようにメソッドの定義を記述するだけで目的コードを構築することができる。

```
CLASS[...
  #コンストラクタのコードなどは補完される
  DEFMETHOD[ ... ],
  ...
]
```

次に、適用例と従来の記述の比較として以下の構文規則で定義されるノードoperate内でのコード生成部の記述例を示す。

```
expr: expr '+' term
      => operate('+', expr, term)
```

我々の中間表現を使わずにJVMコードを構築する場合、この生成規則に対する処理は次のようになる。まずオペランドの型情報を比較し、その式の型を決定する。そしてもしオペランドの型が揃っていない場合には、そのオペランドの型から式の型に変換するJVMの命令を構築する。次に演算子の種類に応じた基本となるJVMの命令を選択し、式の型に応じた実際のJVMの命令を演算コードとして構築する。そして最後に構築されたオペランド・コード、演算コードの順で全体のコードを構築する。この処理をRubyで実直に記述した場合、約20行の記述が必要となる。

一方本研究の中間表現として定義した要素OOPを用いた場合、上記の処理は以下のような記述となる。

```
#式の型の決定
etype = opl1.etype > opl2.etype ?
 opl1.etype : epl2.etype

#コードの構築
code = OOP[sym, [opl1.code, opl1.etype],
 [opl2.code, opl2.etyp] ]
```

このように本研究のモジュールを用いた場合、先のような処理の記述は変換モジュール内に吸収され、利用者は上記の2行程度の記述を行えば良いだけである。

本研究では目的コードとして、比較的高水準なJavaバイトコードを選択したため、劇的な記述量の削減という結果は得られなかった。しかしモジュールを用いない場合に比べると、ターゲットマシン固有の記述などを行う必要がなく、マシンに関する知識があまりない状態でもコード生成部の作成が比較的容易にできるようになった。

## 5 まとめ

本研究では、コード生成部を手軽に作成できるモジュールの提供を目指した。この中間表現モジュールを用いることで、かなり容易にJVMをターゲットとしたコード生成部の実装ができるようになった。しかし、最適化や生成するコードの効率に関してはほとんど考慮されておらず、これらは今後の課題である。

## 参考文献


[1] Sun Microsystem. The Java Virtual Machine Specification. http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

[2] 舞田純一, 佐藤聡, 中井央. 拡張可能なコンパイラ生成系. 情報処理学会論文誌：プログラミング, Vol. 47, No. 16, pp. 1–9, 2006.

[3] 中田育男. コンパイラ. オーム社, 1995.

[4] Igarashi. A., Pierce, B. and Wandler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ, Proc. 1999 ACM *Transactions on Programming, Languages and Systems*, Vol.23, No.3, pp.396-450, May 2001.