平成23年度 学士論文

HPC向け実行時言語変換器 WootinJのバイトコード対応と 表現力の拡張

東京工業大学 理学部 情報科学科
学籍番号 08-2175-4
穂積 俊平

指導教員
千葉 滋 教授

平成24年2月7日

概要

本研究では、HPC 向け実行時言語変換器 WootinJ に対し、変換にかかるオーバーヘッドの削減と表現力の拡張の２つの改善をおこなった。WootinJ とは、実行速度とモジュラリティを両立することを目的とした Java から CUDA への言語変換器である。WootinJ のユーザは、Java の高いモジュラリティを利用してコードを記述することができ、かつ、記述されたコードは CUDA に変換され、高速に実行される。WootinJ はより高速実行可能なコードを生成するために、実行時の情報を利用することで、最適化をおこなっている。

１つ目の改善点は、変換にかかるオーバーヘッドを削減したことである。WootinJ は実行時に変換をおこなうため、変換にかかるオーバーヘッドが実行時間に加算されてしまう。そのため、変換にかかるオーバーヘッドはなるべく小さくする必要がある。WootinJ は、JastAddJ を利用して Java ソースコードから Java AST を構築、構築した Java AST から CUDA コードを生成している。このうち、オーバーヘッドの原因となっているのは、JastAddJ を利用した Java AST の構築である。JastAddJ は静的に利用されることを想定した Java コンパイラであり、Java AST の構築に長い時間をかけている。そこで、Java AST の構築にかかる時間を削減するために、Java バイトコードから Java AST を構築する新しい手法を実装した。

２つ目の改善点は、HPC で有用なデータ構造を利用可能にしたことである。HPC では、膨大なデータや巨大な値が使用される。そのため、長さが long で表現された配列や unsigned long などのデータ構造は有用だと考えられる。しかし、WootinJ でこれらのデータ構造を利用することは困難であった。WootinJ の入力である Java には、これらのデータ構造に対応する明確な表現が存在しないため、WootinJ を利用した際に、これらのデータ構造を利用したようなコードを生成することができないためである。しかし、WootinJ の変換先の言語である CUDA では、これらのデータ構造はプリミティブに提供されているため、Java 上の表現が確立されれば、これらのデータ構造を利用することができる。これを可能にするために、特殊なアノテーションの導入をおこなった。導入したアノテーションを利用することで、クラス、メソッドに対し、変換後の表現を明確に指

定することができ、結果として、これらのデータ構造を利用することができるようになる。

本研究では、旧来の手法と新しい手法で Java AST を構築する時間を測定し、新しい手法の方が実行時間が短いことを確認した。また、導入したアノテーションを利用し、長さが long で表現された配列を利用できることを確認した。

謝辞

本研究を進めるにあたり、研究の方針や論文の構成法など数々の有用な助言を頂いた指導教員の千葉滋教授に心より感謝いたします。また、本研究の仕様、実装、実験方法など多岐にわたり助言をしてくださった伊尾木将之氏に大変感謝いたします。最後に、情報科学に関する多くの知識を与えてくださった千葉研究室の皆さんに御礼を申し上げます。

目 次

図 目 次

表 目 次

第1章　はじめに

高性能計算 (HPC) のプログラムは、より良いパフォーマンスを得るために、スーパーコンピューターなどのハードウェアを意識して記述されてきた。しかし、近年ハードウェアを意識したプログラミングは難しい作業となっている。スーパーコンピュータにマルチコア、コンピュータ・クラスタ、GPGPU など多様なアーキテクチャが取り入れられたことにより、プログラマーは以下に挙げるようなパフォーマンスに関するコードを記述する必要が出てきたためである。

- 並列化
- データ転送
- メモリバンド幅の最大化

これらのパフォーマンスに関するコードはプログラム保守の観点から、本来の計算に必要なコードとは分離されるべきである。しかし、現状の HPC のプログラムでは、以下に挙げる理由からこれらは混在してしまっている。

- HPC のプログラムで使われるプログラム言語は速度を重視しており、抽象化能力が低い。
- 提供されているライブラリはパフォーマンスチューニングを行えるように、低級な操作を提供しているため、プログラマーは明示的にパフォーマンスに関するコードを記述しなければいけない。

パフォーマンスに関するコードと計算に関するコードを分離するためには、Java などの抽象化能力の高いプログラム言語を導入すればよいと考えられる。しかし、これらの抽象化能力の高い言語は、Fortran などの HPC で利用されてきた言語と比較すると、一般的に低速である。そのため、実行速度が重要視される HPC のプログラムにこれらの言語を導入することは難しい。

この問題を解決するために、抽象化能力の高い言語から実行速度の速い言語への言語変換器が多数開発されている。本研究では、それらの言語変

換器のうちの 1 つである WootinJ に着目した。WootinJ は、実行速度とモジュラリティを両立することを目的とした Java から CUDA [8] への言語変換器である。WootinJ を利用することで、ユーザは抽象化能力の高い Java でプログラムを記述しつつ、高速に実行することができる。また、WootinJ では、記述できる内容に一定の制限を設けることで、高速実行可能なコードを生成している。

本稿では、WootinJ の問題点を指摘し、それらをいかに解決したか述べる。WootinJ の問題点とは、以下の 2 つである。

- 変換にかかるオーバーヘッドが大きい
- HPC で有用なデータ構造の利用が困難である

これらを解決するために、Java バイトコードからの Java AST 構築、特殊なアノテーションの導入を行った。

本稿の残りは、次のような構成になっている。第 2 章で、HPC を目的としたプログラミングの現状と、WootinJ の概要、問題点を説明し、第 3 章で、WootinJ の問題に対する改善策を説明する。第 4 章では、改善策が WootinJ にもたらした効果を実験を通して検証する。そして第 5 章でまとめと今後の課題を述べる。

第2章　HPCを目的としたプログラミングとWootinJ

この章では、まずHPCを目的としたプログラミングの現状を述べ、その問題点と問題点に対する解決方法を示す。その後、解決方法の1つである実行時言語変換器WootinJについて説明をおこない、その問題点を指摘する。

2.1　HPCを目的としたプログラミング

HPC(High Performance Computing、高性能計算)とは、単位時間あたりに多くの計算をおこなうことである。狭義には、並列計算機やスーパーコンピュータを用いて、大規模な問題を計算することをいう。本稿では、後者の意味でHPCを使用する。この節では、HPC向けのプログラムで使用されるプログラミング言語と、プログラムが実行される環境について説明をおこなう。

2.1.1　プログラミング言語

HPCを目的としたプログラミングでは、Fortranがよく利用されている。Fortranは、「数式翻訳」を意味する英語「formula translation」が由来であり、数値計算を目的として開発されたプログラミング言語である。そのため、各種組込み関数、複素数、強力な配列演算など数値計算において便利な機能が多数取り入れられている。また、Fortranは、文法がシンプルなため、コンパイラによる最適化がおこないやすく、実行速度の面で他の言語より優れていると言われる。

- 複素数
  Fortranでは、言語レベルで複素数をサポートしている。複素数を表すcomplex型が存在し、簡単に複素数を扱うことができる。例えば、a + biという複素数を扱いたい場合、以下のように記述することで、変数cに、a + biという値を保存することができる。このように宣言した複素数の変数に対して演算を行うこともできる。

complex :: c = (a,b)

- 配列
  Fortran では、多次元配列をサポートしている。例えば、

  integer,dimension(2,2,2)

  という記述によって、整数型の $2 \times 2 \times 2$ の配列を宣言することができる。また、配列に対する強力な演算を多数備えており、配列全体に対して代入や算術をおこなうことができる。例えば、配列 a に対して一括代入を行う場合、以下のように書くだけでよい。

  a = 10

  また、配列 a に対して、すべての要素に 1 を足したい場合、以下のように書くだけでよい。

  a = a + 1

2.1.2　実行環境

HPC 向けのプログラムは並列計算機やスーパーコンピュータによって実行されている。近年のスーパーコンピュータには、より高い性能を実現するために、様々なアーキテクチャが取り入れられている。ここでは、それらのうち代表的なものを取り上げ説明をおこなう。

マルチコア

マルチコアとは、複数のプロセッサ・コアを 1 つのプロセッサ・パッケージに集積したマイクロプロセッサである。各プロセッサ・コアは OS からは複数のマイクロプロセッサとして扱われる。

コンピュータ・クラスタ

コンピュータ・クラスタとは、複数のコンピュータを結合し、葡萄の房 (クラスタ) のようにひとまとまりにしたシステムのことである。複数のコンピュータを結合することで、一台のコンピュータでは得られない高い処理能力を実現することができる。また、大容量のメモリを必要とする計算をおこなうのにも適している。

GPGPU

GPGPU(General-purpose computing on graphics processing units) とは、GPU(graphics processing units) を画像処理だけではなく、より汎用的な計算に用いることである。GPU とは一般的に画像処理をおこなうプロセッサである。GPU はシンプルな演算ユニットを多数搭載しており、並行性の高い演算処理をおこなう場合、CPU よりも高い性能を発揮することができる。

2.2　HPC 向けプログラムの問題点

HPC 向けプログラムが持つ問題点は、計算に関するコードとパフォーマンスに関するコードが混在していることである。前節で述べたように、HPC 向けのプログラムが実行されるスーパーコンピュータなどのハードウェアは、近年多様なアーキテクチャを取り入れている。それに伴い、並列化などパフォーマンスに関するコードを、プログラマーは記述しなければならなくなった。これらのパフォーマンスに関するコードと、本来の計算に必要なコードは、プログラム保守の観点から分離されるべきである。しかし、HPC を目的としたプログラミングにおいて、これらのコードを分離することは難しい状態にある。なぜならば、HPC を目的としたプログラミングで用いられるプログラミング言語の抽象化能力が低いためである。また、プログラミングを支援するために、MPI [7] などのライブラリが提供されているが、これらはパフォーマンスチューニングがおこなえるように低レベルな操作を提供している。そのため、これらを利用したとしても、プログラマーはパフォーマンスに関するコードを記述する必要がある。

ここからは、HPC 向けプログラムの問題点の原因となっているプログラミング言語、プログラミング支援について詳しく説明する。

2.2.1　プログラミング言語

前述したように、HPC では Fortran が長年にわたって利用されている。Fortran は長い歴史を持つ言語で、最初の Fortran が開発されたのは、1957 年である。Fortran には開発されてから現在まで、多数の機能が追加されてきた。例えば、1958 年に開発された FORTRANNII では、手続き型プログラミングがサポートされ、関数を定義できるようになった。また、1991 年に開発された Fortran90 では、構造化プログラミングが取り入れられ、

よりわかりやすいコードを記述できるようになった。Fortran90 では、構造体などのサポートも行われた。このように、様々な機能が追加され、進化してきた Fortran であるが、近年に開発された他のプログラミング言語と比較すると、十分な抽象化能力を持っているとは言いがたい。例えば、Fortran を用いてオブジェクト指向プログラミングをおこなうことは困難だと考えられる。構造体を利用すれば、オブジェクト指向プログラミングをおこなうことは可能だが、複雑なプログラムになると考えられる。一方、Java などの近年開発された手続き型言語の多くは、オブジェクト指向をおこなうために便利な機能を提供している。例えば、Java では、class を用いることで、簡単にオブジェクト指向プログラミングをおこなうことができる。このように抽象化能力の低い Fortran であるが、前述した通り数値計算に便利な機能が多数備わっている、ライブラリの莫大な遺産があるなどの理由から、HPC では Fortran がよく利用されている。HPC を目的としたプログラミングでは、抽象化能力の低い Fortran が利用されているため、適切なモジュール分割をおこなうことが難しくなっている。

2.2.2　プログラミング支援

HPC を目的をしたプログラミングを支援するために、いくつかのライブラリやプログラミング言語が開発されている。それらは、前提とするメモリアーキテクチャによって 2 つにわけることができる。

- 共有メモリモデル
  共有メモリモデルとは、複数のプロセッサが単一のメモリを共有するメモリモデルである。近年のスーパーコンピュータに搭載されている CPU は上に述べたように、マルチコア化されている。そのため、複数のプロセッサが単一のメモリを共有するような状況が生まれている。共有メモリモデルの概念を表したのが図 2.1 である。

- 分散メモリモデル
  分散メモリモデルとは、プロセッサとメモリが一対一で対応しており、それらが何らかの通信手段で接続されたメモリモデルである。1 つのプロセッサとメモリの対をノードを呼ぶ。スーパーコンピュータがコンピュータ・クラスタの形態をとるときに、このようなメモリモデルとなる。分散メモリモデルの概念を表したのが図 2.2 である。

このうち、プログラミングをより複雑にしているのは、分散メモリモデルである。分散メモリモデルに対するライブラリとして、MPI [7] ライブラリが広く利用されている。ここからは、MPI ライブラリについて詳し

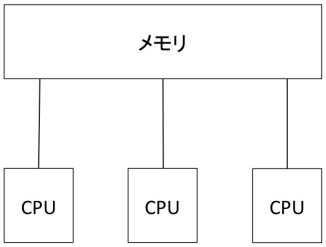

図 2.1: 共有メモリモデル

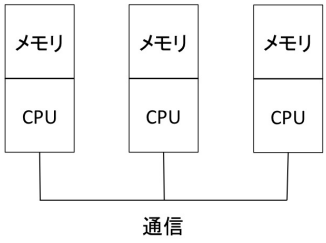

図 2.2: 分散メモリモデル

く説明し、プログラマーがデータ転送を明示的に記述しなければいけないことを示す。

MPI

MPI(Message Passing Interface) とは、メッセージ通信 API の規格である。MPI ライブラリは、主に分散メモリモデルにおいて、ノード間のデータ転送に使用される。分散メモリモデルでは、各ノードのローカルメモリにデータが保存されるため、他ノードのデータが必要な場合、通信を行いデータを取得する必要がある。この通信に利用されるのが、MPI ライブラリである。

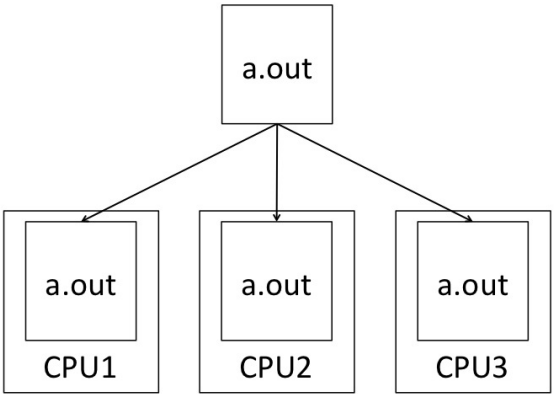

図 2.3: SPMD モデル

MPI ライブラリを用いたプログラムは、SPMD(Single Program Multi Data) と呼ばれるプログラミングモデルで動作する。SPMD とは、1 つのプログラムで並列処理を記述する方式であり、図 2.3 のように 1 つのプロ

グラムが複数のプロセッサで動作する。1 つのプログラムで、各プロセッサ毎に処理を分散させるためには、各プロセッサに対応する ID が存在すればよい。MPI ライブラリでは、システムから、各プロセッサに RANK という ID が付与される。プログラマーは、この RANK を用いて、各プロセッサ毎に異なる処理をおこなわせるプログラムを記述することができる。図 2.4 は RANK を用いたコード例である。size がプロセッサの総数を、rank が各プロセッサに対応する RANK を表している。

```
1 call mpi_init(err)
2 call mpi_comm_size(MPI_COMM_WORLD, size, err)
3 call mpi_comm_rank(MPI_COMM_WORLD, rank, err)
4 print *, 'size=', size, 'rank=', rank
5 call mpi_finalize(err)
6 end
```

図 2.4: RANK を利用したコード例

このプログラムを 3 つのプロセッサで並列に実行すると、以下のような結果を得ることができる。

size=3 rank=0
size=3 rank=1
size=3 rank=2

MPI を用いたプログラミング例として、次のような 1 次元配列 (長さ 9 とする) の計算を考える。

$$n'(i) = n(i\text{-}1) + n(i)$$

この計算を、配列を 3 つに分割し、並列的におこなうことを考える。図 2.5 は、この分割の様子を示した図である。3 つに分割するため、各 RANK(0 ， 1 ， 2) における配列の長さは 3 となる。しかし、それぞれの配列が持つ情報のみでは計算をおこなうことはできない。配列の両端の値を計算するためには、その隣の値が必要となるが、その値は別の RANK に属する配列が保持している。例えば、RANK 0 の 3 番目の値の計算をおこなうためには、RANK 1 の配列が持つ 4 番目の値を取得する必要がある。そこで、MPI ライブラリが提供する関数を用いてデータ転送をおこない、配列の両端の隣の値を取得する。また、各配列の長さを 5 とし、計算に必要な値を格納できるようにする。配列の長さを 5 とする様子は、図 2.5 において、灰色で示されている。

MPI ライブラリでは、RANK 間の通信をおこなうための関数群を提供している。MPI ライブラリが提供する関数のうち、mpi_isend、mpi_irecv、

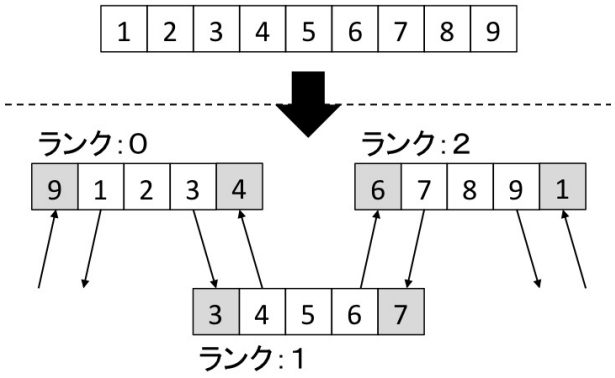

図 2.5: 配列の分割と分割された配列間のデータ転送のイメージ

mpi_wait を使い、この例の計算を実装したのが図 2.6 である。ただし、MPI を使うための初期化コードや、配列などのローカル変数の初期化コードは省略している。

この例に見るように、MPI ライブラリを使ったプログラミングでは、明示的に RANK 間でのデータ転送に関するコードを記述する必要がある。プログラマーがデータ転送に関するコードを記述するということは、データ転送のタイミングを変更することができることを意味する。そのため、データ転送のタイミングを最適化することで、プログラムのパフォーマンスを向上させることができる。しかし、データ転送に関するコードを記述しなければいけないことで、本来の計算に必要なコードとの混在という問題を生じている。

```
1    leftRank = myRank - 1
2    if (leftRank == 0) leftRank = size - 1
3    rightRank = myRank + 1
4    if (rightRank == size - 1) rifhtRank = 0
5
6    call mpi_isend(n(length),1,MPI_INTEGER,rightRank,1,
7                                   &MPI_COMMON_WORLD,req ,err)
8    call mpi_irecv(n(0)      ,1,MPI_INTEGER,leftRank ,1,
9                                   &MPI_COMMON_WORLD,req1,err)
10   call mpi_wait(req)
11   call mpi_wait(req1)
12
13   call mpi_isend(n(1)        ,1,MPI_INTEGER,leftRank ,1,
14                                    &MPI_COMMON_WORLD,req2,err)
15   call mpi_irecv(n(length+1),1,MPI_INTEGER,rightRank,1,
16                                    &MPI_COMMON_WORLD,req3,err)
17   call mpi_wait(req2)
18   call mpi_wait(req3)
19
20   do i = 1,length
21     new_n(i) = n(i-1) + n(i) + n(i+1)
22   end
```

図 2.6: MPI を用いたプログラムにおける通信の例

2.3　HPC 向けプログラムの問題点に対する解決策

HPC 向けプログラムの問題点は、計算に関するコードとパフォーマンスに関するコードが混在していることだと述べた。これらのコードを分離するためには、Java のような抽象化能力の高いプログラミング言語を導入すれば良いと考えられる。しかし、Java のように抽象化能力が高い言語は、Fortran などのこれまで HPC で利用されてきた言語と比較すると、実行速度が遅いと言われている。そのため、実行速度が重要視される HPC に、Java のような抽象化能力の高い言語を導入することは難しい。そこで、抽象化能力の高い言語から実行速度の速い言語への変換器がいくつか開発されている。これらの変換器を利用することで、ユーザは抽象化能力の高い言語でプログラムを記述しつつ、プログラムを高速に実行することができる。以下に、言語変換器の例を示す。

- Aprapi [1]
  Aparapi は、Java から OpenCL への言語変換器である。Pure Java のみで記述ができ、Pure Java で記述されたコードは、OpenCL へと変換され実行される。もし、OpenCL に変換ができない場合は、

Java コードとして実行される。変換されるコードに記述できる内容には制限があり、基本的にユーザはプリミティブな配列を用いた C スタイルのコードしか記述できない。オブジェクトは許されず、try-catch-finally ブロックも許されず、配列の割り当て・解放もできない。

- Jconqurr [3]
  並列計算を行うためのツールである。ループ構造を CUDA [8] に変換し、並列実行することができる。Jconqurr が提供するアノテーションをループ構造に付与する。付与されたループ構造は、Jconqurr によって CUDA コードに変換され、GPU で実行される。

- Toba [10]
  Java から C 言語への言語変換器である。静的に Java バイトコードから C 言語への変換をおこなう。制御構造の復元は行なっておらず、変換後のコードには、goto 文が含まれる。

- WootinJ
  Java から CUDA への言語変換器である。HPC 向けのプログラミングにおいて、実行速度とモジュラリティを両立することを目的とする。Pure Java のみで記述可能であり、オブジェクトやダイナミックメソッドディスパッチの使用も可能である。生成されるコードは実行時の情報を用いて最適化されている。ただし、最適化を行うために、記述できるコードには一定の制限が課せられる。

本研究では、これらのうち WootinJ に着目し、WootinJ をより改善する方法を考え実装した。ここからは WootinJ について詳しく説明し、その後、現状の WootinJ の問題点を指摘する。

2.4　WootinJ

WootinJ は、HPC を目的としたプログラミングにおいて、実行速度とモジュラリティを両立することを目的とした実行時言語変換器である。WootinJ は、Pure Java で記述されたコードを、CUDA コードに変換し、実行する。そのため、WootinJ のユーザは、抽象化能力の高い Java でプログラムを記述しつつ、プログラムを高速に実行することができる。WootinJ はメソッド単位で、コード変換をおこなっている。この際に、メソッドのレシーバと実引数が与えられる。WootinJ はこのレシーバと実引数の型を利用し、コードに最適化を施している。ただし、この最適化を行うため

に、記述できるコードに一定の制限を設けている。

ここからは、まず WootinJ の変換先の言語である CUDA について説明をおこない、その後、WootinJ について詳細な説明をおこなう。

2.4.1　CUDA

CUDA とは、NVIDIA が提供する GPGPU を目的とする、C 言語を拡張したプログラミング言語である。CUDA のプログラムは、CPU で実行されるホストコードと、GPU で実行されるデバイスコードから構成される。ホストコードから呼び出されるデバイスコードの関数をカーネルメソッドと呼ぶ。CUDA は、グリッド、ブロックという概念でスレッドの管理をおこなっており、カーネルメソッドの呼び出しの際には、これらを指定する必要がある。

2.4.2　WootinJ 実行系

WootinJ を用いた Java から CUDA コードへの変換は、メソッド単位でおこなうことができる。WootinJ を利用するユーザは、あらかじめ CUDA コードで実行してほしい計算を特定のメソッドに記述しておく。WootinJ が提供する CUDARunner クラスの invoke メソッドを用いることで、ユーザは指定したメソッドを GPU で実行することができる。ここからは invoke メソッドによって実行されるメソッドを、CUDA の命名にならい、カーネルメソッドと呼ぶこととする。invoke メソッドの概要は以下の通りである。

CUDARunner.invoke(Object obj, String method, Object... args)

- obj
  メソッドのレシーバとなるオブジェクト
- method
  呼び出すメソッドの名前
- args
  メソッドに渡す実引数

invoke メソッドの呼び出しが行われると、WootinJ は以下の処理をおこなう。

1. obj の型、method、args の型を利用して、変換の対象とすべきメソッドを特定し、そのソースコードを取得する。
2. JastAddJ [2] を用いて、取得したソースコードから Java AST(抽象構文木) を構築する。
3. 構築した Java AST から CUDA コードを生成する。
4. nvcc を用いてコンパイルを行い、共有ライブラリを作成する。
5. JNI を利用して、作成した共有ライブラリを実行する

WootinJ は、ユーザが指定したメソッドをカーネルメソッドとして呼び出すような CUDA コードを生成する。カーネルメソッド呼び出しの際には、スレッド数を指定する必要があるが、これは WootinJ が提供する MethodConfig を用いることで指定することができる。

カーネルメソッド内で利用されるデータは、WootinJ によって自動的に Java メモリから GPU メモリへと転送される。GPU メモリに送られたデータに対し変更があった場合、Java メモリ上のデータにその変更は自動的に反映されない。GPU メモリ上のデータを Java メモリ上のデータに反映させるためには、WootinJ が提供する CUDAData クラスの get メソッドを利用する必要がある。このように、ユーザが明示的にデータ転送を行わなければいけないシステムにしているのは、データ転送の回数を極力減らし、オーバーヘッドを削減するためである。

2.4.3　動的メソッドディスパッチの除去

WootinJ は、invoke メソッドの引数として、カーネルメソッドのレシーバと実引数を得ることができる。これらの実行時の型を利用することで、カーネルメソッド内における、レシーバのフィールドと実引数に対するメソッド呼び出しを、静的に解決したコードを生成している。今、図 2.7 中に示された kernel という名前のメソッドが、変換の対象となるメソッドであるとする。本来であれば、kernel 中の hoge.get() は hoge の実行時の型を用いて、動的に解釈される。しかし、WootinJ を用いて変換することで、hoge.get() の箇所が静的に解決されたコードを得ることができる。hoge の実際の型が Hoge であった場合、図 2.8 のコードを得ることができる。また hoge の実際の型が Piyo であった場合は、図 2.9 のコードを得ることができる。

WootinJ は、カーネルメソッド内の全てのメソッドを静的に解決したコードを生成する。先に説明したように、レシーバのフィールドと実引数に対するメソッド呼び出しは、それらの実際の型を利用することで静的に

```
1  class Hoge {
2    int get(){
3      return 1;
4    }
5  }
6
7  class Piyo extends Hoge {
8    int get(){
9      return 2;
10   }
11 }
12
13 void kernel(Hoge hoge){
14   int num = hoge.get();
15 }
```

図 2.7: 動的メソッドディスパッチが利用された Java コード例

```
1  int Hoge_get(){
2    return 1;
3  }
4
5  void kernel(){
6    int num = Hoge_get();
7  }
```

図 2.8: 図 2.7 の Java コードにおいて引数の実際の型が Hoge であった場合に生成される CUDA コード

解決したメソッド呼び出しへと変換できる。それ以外のカーネルメソッド内に記述されたメソッド呼び出しは、すべて静的に解決できることが要求される。これを満たすために、WootinJ が要求する制限は以下の通りである。

- 代入式の両辺は strict-final 型である
- カーネルメソッドの返り値が strict-final 型である。
- 配列が strict-final 型である。

strict-final 型に求められることは、実際の型と宣言された型が等しいことである。つまり、strict-final 型の要素に対するメソッド呼び出しは静的に解決することができる。strict-final 型の条件は以下の通りである。

```
1  int Piyo_get(){
2    return 2;
3  }
4
5  void kernel(){
6    int num = Piyo_get();
7  }
```

図 2.9: 図 2.7 の Java コードにおいて引数の実際の型が Piyo であった場合に生成される CUDA コード

- primitive 型は strict-final 型である。
- primitive 型を要素とする配列は strict-final 型である。
- final かつフィールドが strict-final 型なクラスは strict-final 型である。

2.4.4　オブジェクト構造の除去

```
1  final class Hoge {
2    int a;
3    Piyo piyo;
4  }
5
6  final class Piyo {
7    int b;
8  }
9
10 int sum(Hoge hoge){
11   return hoge.a + hoge.piyo.b;
12 }
```

図 2.10: オブジェクトが利用された Java コード例

WootinJ は、オブジェクトをプリミティブな値もしくはそれを格納する変数に置き換えた C コードを生成している。オブジェクトを C コード中に表現するには、構造体を使った方法も考えられる。構造体を用いてオブジェクトを表現した場合、フィールドアクセスはアロー演算子を用いて表現されることとなる。しかし、アロー演算子のアクセスには多少のオーバーヘッドがかかる。そのため、より高速実行可能な C コードを生成する

```
1  int sum(int Hoge_a, int Hoge_piyo_b){
2    return Hoge_a + Hoge_piyo_b;
3  }
```

図 2.11: 図 2.10 からオブジェクト構造を除去し、生成された C コード

ために、WootinJ はオブジェクトをプリミティブな値もしくはそれを格納する変数に置き換えることで表現している。例として、図 2.10 と図 2.11 を示す。図 2.10 に示された sum メソッドに対し、図 2.11 の変換結果を得ることができる。オブジェクトに対しこのような変換をおこなうために、WootinJ はユーザに対し以下の制限を設けている。

- オブジェクトの型が再起型でない
- オブジェクトのエイリアスが作られたあとに、元のオブジェクトとエイリアスに対し副作用がない
- もし、変換の対象となるオブジェクトが、カーネルメソッドの引数であった場合、そのオブジェクトに対して副作用がない

オブジェクトの型が再起型でないとは、オブジェクトのフィールドにそのオブジェクトの型を持つものが存在しないことを意味する。もし、オブジェクトが再起型であるとすると、オブジェクトをプリミティブ型の値へ変更する際に、そのオブジェクトのフィールドを特定することができず、無限ループに陥ってしまう。現在の WootinJ では、エイリアスに対する解析を行なっていない。そのため、エイリアスに対する変更をもとのオブジェクトに反映することができない。また、逆も同様である。したがって、カーネルメソッド内のオブジェクトに対し副作用がないことが要求される。ここで、副作用とはオブジェクトやオブジェクトのフィールドに対する代入を意味する。

2.5　WootinJ の問題点

現状の WootinJ には、HPC 向けのプログラムで利用するにあたって、問題点がある。この節では、現状の WootinJ が抱える問題点について述べる。WootinJ が持つ問題点は大きく分け 2 つあると考えている。

- 変換にかかるオーバーヘッドが大きい
  WootinJ の持つ 1 つ目の問題点は、Java AST の構築に長い時間を要している点である。WootinJ は受けとったメソッド情報を用いて Java

ソースコードを取得し、取得した Java ソースコードから JastAddJ を用いて Java AST を構築している。しかし、この方法では、Java AST の構築に多くの時間を要してしまう。なぜなら、JastAddJ は時間をかけて Java AST を構築するためである。JastAddJ は Java を拡張した新しい言語を作成する場合などに用いられる Java コンパイラであり、本来静的に利用されることを想定しているため、時間をかけて Java AST を構築する。Java AST を構築するのに長い時間を要することは問題である。WootinJ は実行時にコードの変換をおこなっているため、Java AST の構築にかかる時間がそのまま実行時間に加算されてしまう。そのため、Java AST の構築に長い時間がかかると、WootinJ のオーバーヘッドが大きくなってしまう。

- HPC で有用なデータ構造の利用が困難
  ２つ目の問題点は、HPC で有用なデータ構造を利用しづらい点である。WootinJ は HPC 向けの言語変換器であり、大量のデータや巨大な値を扱うプログラムが WootinJ を用いて実行されることが想定される。したがって、大量のデータを扱うために長さが long で表現された配列を用いたり、巨大な値を扱うために unsigned long に相当する値が利用されることが考えられる。しかし、Java を記述言語とする WootinJ ではこれらのデータ構造を利用することは難しい。Java では、配列の長さは int 型で表現されることが仕様で決められている。そのため、長さが long の配列を表現しようとした場合、複数の配列を用いて、実装をおこなう必要がある。また、Java の配列アクセスは、毎回配列のサイズ内であるかのチェックをおこなうため、C 言語などの配列アクセスに比べると低速だと考えられる。Java において unsigned long に相当する値を利用するには、BigDecimal クラスを利用する必要がある。しかし、WootinJ では、オブジェクトをプリミティブな型の使用に置き換えるため、BigDecimal クラスを利用するのは困難だと考えられる。

第3章　WootinJのバイトコード化と表現力の拡張

本研究では、前章で述べたWootinJの問題点を改善した。この章では、WootinJの問題点を解決するために、WootinJをどのように改善したかを述べる。最初に、Javaバイトコードから Java ASTを構築するように変更したことを述べ、その後で、HPCで有用なデータ構造を使いやすくするために、特殊なアノテーションを導入したことを述べる。

3.1　Javaバイトコードからの Java AST構築

この節では、Javaバイトコードを解析し、Java ASTを構築するように変更したことを述べる。最初に、Javaバイトコードを入力として選んだ理由を説明し、その後Javaバイトコードから Java ASTを構築するアルゴリズムの説明をおこなう。

3.1.1　Javaバイトコードを入力とした理由

WootinJでは、JastAddJを用いて Java ASTを構築していたために、Java ASTの構築に長い時間を必要としていた。そこで、Java ASTを構築する時間を削減するために、JastAddJを用いずに Java ASTを構築する方法を考え実装し、WootinJに組み込んだ。Java ASTを構築するにあたって、入力として、Javaソースコード、Javaバイトコードの二つの選択肢があったが、Javaバイトコードを選択した。Javaバイトコードとは、Javaのコンパイラが生成する中間コードのことである。Javaのソースコードは Javaバイトコードと呼ばれる中間表現に変換され、クラスファイルと呼ばれる形式で保存される。Javaのプログラムは、このクラスファイルで配布される。そのため、プログラムの実行中に、常に Javaソースコードは入手可能とは限らない。一方、Javaバイトコードは、確実に入手することができる。また、Javaバイトコードを入力とすることで、より多くの言語を選択することが可能となる。Javaバイトコードに変換され、Java仮想機械上で動作する言語は Javaだけではない。Scala [9]、AspectJ [6]、

JRuby [5]、Groovy [4] などの多くの言語が開発されている。Java バイトコードを入力とすることで、WootinJ のこれらの言語への応用が容易となる。

3.1.2　Java AST 構築のアルゴリズム

Java バイトコードから Java AST を構築するためには、デコンパイルに相当する処理をおこなう必要がある。そこで、Java のデコンパイルに関する論文である丸山 [15] を参考とした。ここからは、丸山 [15] のアルゴリズムの概要と、アルゴリズムのうち論文中で詳しく述べられていない箇所についての説明をおこなう。

Java AST を構築するために、丸山 [15] の手法では、コントロールフローグラフと支配木を利用している。そこで、丸山 [15] のアルゴリズムの概要を説明するにあたり、まず、コントロールフローグラフと支配木について説明をおこなう。

コントロールフローグラフ

コントロールフローグラフとは、プログラムにおける制御の流れをグラフで表現したものである。プログラムの分岐や合流の様子を示しており、分岐も合流もない部分を節点として、これらの間を分岐や合流を表す有向辺で結んだ有向辺グラフである。図 3.2 は、図 3.1 のソースコードに対応したコントロールフローグラフである。

```
1   public void makeThesis(){
2     prepare();
3     while(!finish()){
4       if(tired()){
5         rest();
6       }else{
7         study();
8       }
9       write();
10    }
11    submit();
12  }
```

図 3.1: プログラム例

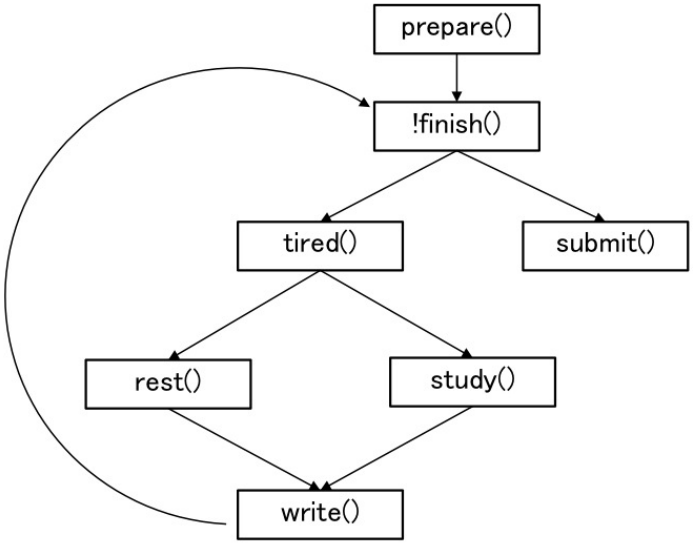

図 3.2: 図 3.1 のプログラムから得られるコントロールフローグラフ

支配木

支配木とは直接支配の関係を辺とした木構造である。直接支配、厳密支配、支配の定義は以下の通りである。

- 支配
  コントロールフローグラフにおける 2 つのノード X と Y について、プログラムの入り口から Y に達するどの路を通っても、必ず X を通過するとき、X は Y を支配する。

- 厳密支配
  　　ノード X が Y を支配し、X ≠ Y であるとき、X は Y を厳密に支配する。

- 直接支配
  ノード X が Y を厳密に支配し、X から Y への路に X 以外に Y を厳密に支配するノードが存在しないとき、X は Y を直接支配する。

図 3.3 は、図 3.2 のコントロールフローグラフに対応した支配木である。プログラムの先頭を表すノードである prepare から!finish へ到達する全ての路は、必ず prepare を通る。よって、prepare は!finish を支配する。また、prepare ≠ !finish であり。!finish を支配するのは prepare のみであるため、prepare は!finish を直接支配する。したがって、prepare と!finish の

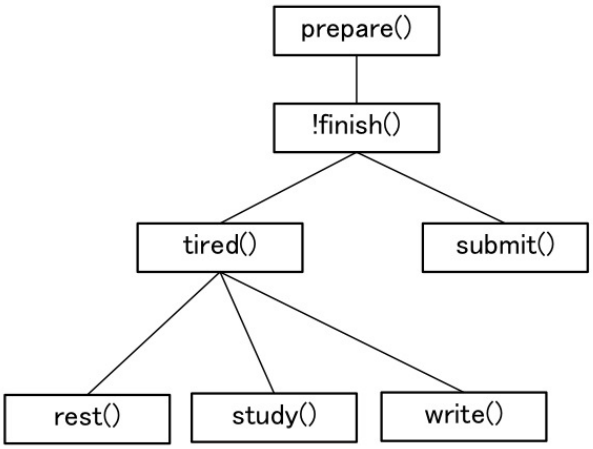

図 3.3: 図 3.2 のコントロールフローグラフに対応する支配木

間に直接支配の関係を表す辺が示されている。少し複雑な例として、study を直接支配するノードを考える。プログラムの先頭を表すノードである prepare から study に達する全ての路は、必ず prepare、!finish、tired を通る。よって、study はこれらのノードに支配される。また、study とは等しくないので、これらのノードは study を厳密に支配する。prepare、!finish は study を支配するが、これらから study への路は、必ず tired を通る。したがって、study への路の途中に、他に study を厳密に支配するノードが存在するため、prepare、!finish は study を直接支配しない。一方、tired から study への路には、他に study を支配するノードは存在しない。よって、tired のみが study を直接支配する。

Java AST 構築のアルゴリズムの概要

では、ここからは、具体的にコントロールフローグラフと、支配木を用いてどのように Java AST を構築しているのか説明する。本研究では、javassist [16] を利用し、コントロールフローグラフと支配木を構築している。全体の流れは以下のようになる。

1. コントロールフローグラフと支配木を利用し、制御構造を復元
2. 復元された制御の流れと支配木を利用し Java AST を構築

丸山 [15] の手法では、支配木を主に用いて、Java AST を構築している。支配木を主に用いる理由は Java AST とそれに対応する支配木の構成が類似しているためである。

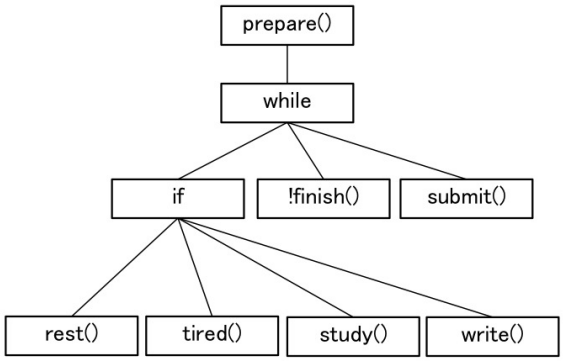

図 3.4: 図 3.1 のプログラムから得られる抽象構文木

図 3.4 は図 3.1 のコードから得られる Java AST である。この Java AST と図 3.3 の支配木を比較すると、類似していることがわかる。具体的には、if-else 文や while などの制御構造が 1 つの部分木をなしている。Java の制御構造に対応するプログラム片は、支配木において、必ず一つの部分木をなすという性質がある。図 3.5 のコードについて考える。(1)cond を評価し、cond が true ならば、(2)B1 を実行する。cond が false ならば、(3)B2 を実行する。この場合、B1 が実行されるのは、cond が ture の場合だけであり、これは (1) の処理が (2) の処理を支配することを表す。同様に、(1) の処理は (3) の処理を支配する。したがって、この if-else 文の制御構造は支配木全体において、一つの部分木をなす。if-else 文以外の制御構造についても同じことが言える。丸山 [15] では、この性質に着目し、支配木を主な解析対象としている。

```
1  if(cond){
2    B1
3  }else{
4    B2
5  }
```

図 3.5: if-else 文例

しかし、支配木だけでは Java AST を構築することはできない。なぜなら、支配木には制御の流れに関する情報は示されていないためである。足りない制御の流れに関する情報はコントロールフローグラフを解析することで得られる。制御の流れはコントロールフローグラフの辺として表現されている。例えば、図 3.3 に示された支配木には while に関する情報は無いが、この支配木に対応したコントロールフローグラフ図 3.2 では、write から!finish への辺として while に関する制御の流れが示されている。丸山 [15] の手法では、コントロールフローグラフの辺として表された制御の流れを、同等の制御構造に置き換えることで、制御構造を解析する。例えば、図 3.3 において、!finish が while の先頭のノードであることを、対応したコントロールフローグラフの情報を元に求める。

3.1.3　制御構造の復元

丸山 [15] でも述べられているが、Java バイトコードから Java AST を構築する際の一番の問題点は、制御構造の復元である。そこで、以下の Java AST 構築のアルゴリズムのうち、1 の制御構造の復元に絞り、より詳細な説明をおこなう。

1. コントロールフローグラフと支配木を利用し、制御構造を復元
2. 復元された制御の流れと支配木から Java AST を構築

制御構造とは、繰り返しや条件判断といった文や命令の実行順序を意味するものである。多くのプログラミング言語は、制御構文と呼ばれる制御構造を記述するための構文を持っている。Java には以下の制御構文があり、Java AST を構築する際にこれらの構造を復元する必要がある。Java バイトコードは、スタックマシンの機械語であるため、制御の流れは、ジャンプ命令 (goto, if, if icmp, if acmp, ifnonnull, ifnull) やメソッド内サブルーチン呼び出し (jsr, ret) だけで表現される。つまり、ソースコードにあったループ、ブロックなどの構造が Java バイトコードでは失われている。そのため、制御構造を復元する必要がある。今回の実装では、これらのうち while、if-else、return に相当する制御構造の復元を行った。

- while
- do-while
- for

- if-else
- switch
- break,continue
- return
- try-catch

丸山 [15] 以外にも制御構造を復元するためのアルゴリズムとして、Tarjan のアルゴリズム [13] や Krakatoa [11] が提案されている。Tarjan のアルゴリズムでは、制御構造を正規表現を用いて表し、一般的に扱うことで制御構造を復元している。しかし、Tarjan のアルゴリズムは入れ子になったループ構造への対応が難しいとされている。また、Krakatoa では、Ramshaw によって提案された Pascal プログラムから goto を取り除くアルゴリズム [12] を拡張して、Java プログラムに適用している。しかし、この方法は、一度不自然な形の Java コードを経るため、実行効率があまりよくないとされる。

丸山 [15] の手法における制御構造を復元するためのアルゴリズムの概要は以下の通りである。ただし、return 文については、Java バイトコードの命令として明確に与えられるため、解析をおこなう必要はない。

1. ループ構造を特定 (while のみ)
   (a) ループ構造のヘッダを特定
   (b) ループ構造の後続ノードを特定
   (c) ループ構造の種類を決定
2. if-else 文を特定
   (a) if-else 文のヘッダを特定
   (b) if-else 文の then 節、else 節を特定
   (c) if-else 文の後続ノードを特定

制御構造を特定するにあたり、それぞれの制御構造のヘッダと後続ノードを求める必要がある。ヘッダとは、それぞれの構造において最初に実行されるノードのことである。図 3.3 の支配木において、while のヘッダとは!finsh をさす。また、後続ノードとは、対象としている制御構造の処理が終わった後に処理が移るノードのことである。図 3.3 において、while の後続ノードとは submit をさす。

ループ構造の特定

ループ構造を特定するにあたって、まずループ構造のヘッダを特定する必要がある。ループ構造のヘッダとは、ループ構造の先頭を表すノードのことである。丸山 [15] では、ループ構造のヘッダの特定方法について詳しく述べられていない。そこで、自分で考え実装した。

ヘッダを特定するために、コントロールフローグラフの解析をおこなった。ループ構造を表すようなコントロールフローグラフには、ある特徴を持った辺が存在することになる。ループ構造を表現するためには、Java バイトコードにおいて、現在の行番号よりも小さい行番号の命令を実行するような分岐命令が必要となる。したがって、この分岐命令の実行を表現した辺がコントロールフローグラフ上に現れることになる。具体的に言うと、行き先のノードの方が、出元のノードよりも Java バイトコードにおける行番号が小さいような辺である。まずは、コントロールフローグラフの解析をおこなうことで、この辺を特定する。この辺の両端のノードうちいずれかがヘッダとなる。どちらがヘッダであるかは、支配木を利用することで判断できる。ループ構造のヘッダは、支配木においてループ構造を表す部分木の根となる。そのため、支配木において、より上位に位置する方のノードがヘッダとなる。

後続ノード、ループ構造の種類の特定 (今回の実装では while のみとしたためおこなっていない) については丸山 [15] において詳しく述べられているため省略する。

if-else 文の特定

if-else 文のヘッダは、ループ構造を特定した後、残された分岐を if-else 文のヘッダとすることで得られる。if-else 文の then 節、else 節、後続ノードの特定方法については、丸山 [15] では詳しく述べられていない。then 節、else 節、後続ノードを特定するために考えたアルゴリズムについて説明をおこなう。

if-else 文の then 節、else 節、後続ノードを特定するにあたって、まず if-else 文の構成について考える。if-else 文は条件式を含むヘッダ、then 節、else 節から構成される。図 3.2 のコントロールフローグラフに含まれる if-else 文において、ヘッダは tired、then 節は rest、else 節は study である。また、この if-else 文の後続ノードは write である。これらの間には先に説明したように、tired が rest、study、write を直接支配するという関係が成り立つ。このように、ヘッダは then 節、else 節、後続ノードを直接支配する。したがって、支配木において then 節、else 節、後続ノードはヘッダの子ノードとなる。

このアルゴリズムでは、支配木において、if-else 文のヘッダが持つ子ノードの個数によって、場合分けをおこなう。このアルゴリズムにおいて、ノードが then 節、else 節のどちらであるかは Java バイトコードの情報を用いて特定が可能である。今回導入したアルゴリズムでは、子ノードの個数が 4 つ以上であるものには対応できていない。しかし、自然なコントロールフローグラフでは子ノードの個数が 4 つ以上になることはほとんどないと考えられる。

1. 子ノードの個数が 3 つの場合
   コントロールフローグラフにおいて、ヘッダから直接の辺が伸びていないノードを後続ノードとする。また、ヘッダから直接の辺が伸びているノードを then 節、else 節をする。

2. 子ノードの個数が 2 つの場合
   コントロールフローグラフにおいて、一方のノード X から他方のノード Y への到達可能性を考える。ただし、ループ構造をまたいだ探索は行わないものとする。

   - 到達する可能性がある場合
     X から Y へ到達する可能性がある場合、Y を後続ノードとする。また、X を then 節もしくは else 節とする。
   - 到達する可能性がない場合
     X,Y をそれぞれ then 節もしくは else 節とする。

子ノードの個数が 3 つの場合、それぞれの子ノードは then 節、else 節、後続ノードのいずれかを表す。この時、コントロールフローグラフにおいて、後続ノードのみヘッダからの辺が存在しない。したがって、ヘッダからの辺が存在しないノードが後続ノードとなる。

子ノードの個数が 2 つの場合、子ノードは then 節、else 節、後続ノードのうちの 2 つを表す。つまり、then 節、else 節、後続ノードのうち、いずれかが存在しない場合と考えられる。後続ノードが存在しない場合、then 節と else 節が存在する。このとき、コントロールフローグラフにおいて、then 節を表すノードから else 節を表すノードへ到達することはできない。(ループを考慮しない) then 節、else 節を入れ替えた場合も同様である。したがって、後続ノードはないと判定できる。後続ノードが存在する場合、then 節もしくは else 節が存在する。then 節、else 節は対称であるため、else 節が存在しないとする。このとき、コントロールフローグラフにおいて、then 節を表すノードから後続ノードへ到達することができるはずである。したがって、到達先のノードを後続ノードと判定できる。

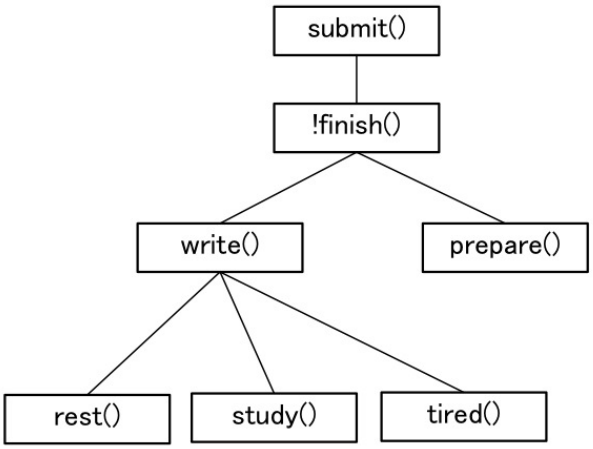

図 3.6: 図 3.2 のコントロールフローグラフに対応する後支配木

後支配木を利用した後続ノードの特定

先に述べたように、制御構造を復元する際には、その制御構造の後続ノードを特定する必要がある。この後続ノードを特定する際に、後支配木を用いることで、効率良く特定ができるのではないかと考えた。後支配木とは、直接後支配の関係を辺とした木構造である。例えば、図 3.2 のコントロールフローグラフに対応する後支配木は、図 3.6 のようになる。後支配、厳密後支配、直接後支配の定義は以下の通りである。

- 後支配
  コントロールフローグラフにおける 2 つのノード X と Y について、Y からプログラムの出口に達するどの路を通っても、必ず X を通過するとき、X は Y を後支配する。

- 厳密後支配
  　ノード X が Y を後支配し、$X \neq Y$ であるとき、X は Y を厳密に後支配する。

- 直接後支配
  ノード X が Y を厳密に後支配し、Y から X への路に X 以外に Y を厳密に後支配するノードが存在しないとき、X は Y を直接後支配する。

具体的には、制御構造のヘッダを直接後支配するノードが存在するならば、そのノードが後続ノードだと考えた。ただし、この方法では、全ての状況で後続ノードを特定できるわけではない。特定できなかった場合については、ループ構造については丸山 [15] の方法を、if-else 文については先に述べた方法を、それぞれ用いて後続ノードの特定を行っている。

3.2　表現力の拡張

２章で述べたように、WootinJ では長さが long で表された配列や unsigned long 型の値を利用するのが困難である。しかし、HPC 向けの言語変換器である WootinJ において、これらのデータ構造は有用だと考えられる。なぜならば、HPC 分野では膨大なデータや巨大な値を利用するためである。そこで、WootinJ において、これらのデータ構造を利用可能にする方法を考え、実装した。

長さが long で表された配列や unsigned long 型の値は、WootinJ の入力である Java では利用することが困難である。だが一方、WootinJ の変換先の言語である CUDA ではプリミティブに提供されている。変換先の CUDA でプリミティブに提供されているにもかかわらず、なぜこれらのデータ構造を利用できないのかというと、これらのデータ構造に対応する Java 上の表現が存在しないためである。そこで、特殊なアノテーションを導入することで、これらのデータ構造に対応した Java 上の表現を作成できるようにした。特殊なアノテーションとは、@type_alias と@fnc_alias である。また、長さが long で表現された配列を GPU メモリ上に確保することは困難だと考えられたため、C コードのみで実行できるようなクラスを WootinJ に追加した。

3.2.1　@type_alias、@fnc_alias

ここからは、@type_alias、@fnc_alias の利用例である BigArrayL クラスを通して、@type_alias と@fnc_alias の説明をおこなう。BigArrayL クラスは要素、長さが long で表された配列を表現したクラスである。このクラスを利用することで、Java では表現が困難な長さが long で表された配列を利用することができる。BigArrayL クラスは、配列に必要な基本的な操作として、以下のメソッドを提供する。

- BigArrayL(long length)
  コンストラクタ
- long get(long index)
  配列の index 番目の要素を返すメソッド

- void set(long index, long value)
  配列の index 番目の要素を value に設定するメソッド

BigArrayL クラスを利用することで、図 3.7 に示すように、自然な形で長さが long で表現された配列を利用することができる。

```
1     static void sample() {
2       long length = 1L << 32;
3       BigArrayL bal = new BigArrayL(length);
4       long i = 0;
5       while(i < length){
6         bal.set(i, i);
7         i++;
8       }
9       long ans = bal.get(100);
10    }
```

図 3.7: BigArrayL クラスを利用した Java コード例

図 3.7 に示したコードは、WootinJ によって、実際には図 3.8 のように変換されて実行される。

```
1     static void sample() {
2       long length = 1L << 32;
3       long* bal = (long*)malloc(length * sizeof(long));
4       long i = 0;
5       while(i < length){
6         bal[i] = i;
7         i++;
8       }
9       long ans = bal[100];
10    }
```

図 3.8: 図 3.7 の Java コードを変換して得られる C コード

図 3.7 のコードと図 3.8 のコードを比較すると、BigArray 型の使用箇所や、BigArray クラスのインスタンスに対するメソッド呼び出し式が変更されていることがわかる。この変更内容は、BigArray クラスのクラス定義において、@type_alias と@fnc_alias を用いて指定されている。図 3.9 は BigArrayL クラスのクラス定義の一部を抜粋したものである。

図 3.9 を見ると、BigArrayL クラスのクラス宣言文に@type_alias が付与されており、引数として long*が渡されている。BigArrayL は要素が long

で表現された配列であるため、C 言語における型は long*である。したがって、@type_alias を使うことで、BigArrayL クラスの C 言語における型が long*であることを定義している。このように、@type_alias を用いることで、Java のクラスに対して、明示的に C 言語上の型を指定することができる。

また、BigArrayL クラスの各メソッドには@fnc_alias が付与されており、引数として C 言語における表現が記述されている。例えば、long get(long index) メソッドに付与された@fnc_alias の引数は、@0[@1] である。C 言語における配列アクセスは [] を用いておこなうことができる。get メソッドに付与された@fnc_alias の引数はそのことを示している。引数に含まれている@0、@1 は、それぞれメソッドのレシーバと引数の一番目を表している。bal.get(100) というメソッド呼び出し式は、bal[100] と変換されることになる。このように、@fnc_alias を用いることで、Java のメソッドに対し明示的に C 言語上の表現を指定することができる。

```
1  @type_alias("long*")
2  public class BigArrayL{
3    @fnc_alias("(long *)malloc(@1 * sizeof(long))")
4    private BigArrayL(long length){
5
6    }
7
8    @fnc_alias("@0[@1]")
9    public long get(long index) {
10     return 0;
11   }
12
13   @fnc_alias("@0[@1] = @2")
14   public void set(long index, long value) {
15
16   }
17 }
```

図 3.9: BigArrayL クラスの定義の一部

@type_alias と@fnc_alias の仕様をまとめると次のようになる。

- @type_alias
  @type_alias は、Java のクラスに対し明示的に C 言語上の型を指定するために利用される。引数には C 言語における型を記述することができる。カーネルメソッドの中で、@type_alias が付与されたクラスが利用されていた場合、そのクラスの利用箇所を@type_alias の引

数に置き換える。現在の WootinJ では対応していないが、コンパイル時に引数の内容が C 言語の型に相当するものかチェックできると考えている。

- @fnc_alias
  @fnc_alias は、Java のメソッドに対し、明示的に C 言語上の表現を指定するために利用される。@fnc_alias は引数として文字列を取る。引数の文字列には、メソッドのレシーバを表す@0 と、対象となるメソッドの引数を表す@n(n >= 1) を含めることができる。カーネルメソッドの中に、@fnc_alias が付与されたメソッドの呼び出し式が記述されていた場合、その呼び出し式を@fnc_alias の引数の文字列に変換する。

3.2.2　CRunner クラス

２章において、カーネルメソッド内で利用されるデータは、WootinJ によって自動的に GPU メモリ上に確保されると述べた。この仕様のもとでは、長さが long で表現された配列を利用するのは困難である。なぜならば、現在の GPU のメモリは多くとも 3GB 程度であるためである。仮に、配列の 1 つ要素が 32bit(4B) で表現され、長さが 2^{31} であるとする。(Java の Integer の最大値は $2^{31} - 1$ である) このとき、配列全体の大きさは約 8GB である。したがって、GPU メモリ上にこの配列を確保することはできない。

この問題を解決するために、WootinJ に C コードを実行するためのクラス CRunner を実装した。C コードで実行することで、CPU 用のメモリを利用することができ、巨大な領域を確保することができるようになる。WootinJ は Java から CUDA への言語変換器であるが、変換先の言語である CUDA は C 言語を拡張した言語である。そのため、変換の対象となるコードに CUDA 固有の表現が含まれない場合、WootinJ は対象となるコードを C コードに変換する。つまり、WootinJ は Java から C 言語への言語変換器だと考えることもできる。

CRunner クラスには、C コード実行するためのメソッド invoke が実装されている。

invoke(Object obj, String method, Object... args)

- obj
  メソッドのレシーバとなるオブジェクト

- method
  呼び出すメソッドの名前

- args
  メソッドに渡す実引数の配列

inovoke メソッドが実行されると、WootinJ は以下の処理をおこなう。

1. obj の型、method、args を用いて、メソッドを特定し、Java バイトコードを取得。
2. 前述した方法で、Java バイトコードから Java AST を構築する。
3. 構築した Java AST から C コードを生成する。
4. gcc を用いてコンパイルを行い、共有ライブラリを作成する。
5. JNI を利用して、作成した共有ライブラリを実行する

ここまで説明した方法で、1 つのカーネルメソッド内において、BigArrayL クラスを利用することができる。ここからは、複数のカーネルメソッドにまたがった BigArrayL クラスの利用を考える。想定しているのは、図 3.10 のように、複数のカーネルメソッドの呼び出しに対して、BigArrayL クラスのインスタンスを渡すことである。

```
1  BigArrayL bal = new BigArray(length);
2  CRunner.invoke(obj, "method1", bal, length);
3  CRunner.invoke(obj, "method2", bal, length);
```

図 3.10: 複数のカーネルメソッドにまたがった BigArraL クラスのオブジェクトの利用

このような実行を可能にするために、CRunner クラスに register メソッドを実装した。

register(Object register, Object obj, String method, Object... args)

- register
  CRunner に登録をおこなうオブジェクト

- obj
  メソッドのレシーバとなるオブジェクト

- method
  CRunner に登録するオブジェクトに対応するポインタを返すメソッド名

- args
  メソッドに渡す引数

register メソッドを利用することで、Java 上のオブジェクトと C 上の配列 (ポインタ) の関連付けをおこなうことができる。register メソッドによって登録されたオブジェクトが、CRunner クラスの invoke メソッドの引数として渡されると、関連付けられたポインタに変換され、カーネルメソッドに渡される。CRunner クラスの register メソッドを利用して、BigArrayL クラスは factory メソッドを提供している。この factory メソッド使って取得した BigArrayL クラスのインスタンスは、図 3.11 のように、複数のカーネルメソッドに対して、引数として渡すことができる。

```
1  BigArrayL bal = BigArrayL.factory(length);
2  CRunner.invoke(obj, "method1", bal, length);
3  CRunner.invoke(obj, "method2", bal, length);
```

図 3.11: factory メソッドを利用した Java コード例

factory メソッドの実装は、図 3.12 のようになっている。factory メソッドが呼び出されると、BigArrayL クラスのインスタンス bal を生成し、bal と make メソッドの返り値として得られたポインタを register メソッドを使い関連付ける。make メソッドは、@type_alias と@fnc_alias によって指定された内容に従い、実際には図 3.13 のように変換され実行される。その返り値は long*型であり、register メソッドはそのポインタと引数として渡された bal を関連付けている。

```
1   public static BigArrayL factory(long length){
2     BigArrayL bal = new BigArrayL();
3     CRunner.register(bal, bal, "make", length);
4     return bal;
5   }
6
7   static BigArrayL make(long length){
8     BigArrayL bal = new BigArrayL(length);
9     return bal;
10  }
```

図 3.12: register メソッドを利用した Java コード例

Listing 3.1: make メソッドの変換後

```
1   long* make(long length){
2     long* bal = malloc(length * sizeof(long));
3     return bal;
4   }
```

図 3.13: 図 3.12 の make メソッドを変換して得られる C コード

第4章　実験

4.1　Javaバイトコード化によるJava AST構築の高速化

Java ASTの構築方法を変更したことで、Java ASTの構築にかかる時間が短縮されたのかを検証するための実験をおこなった。実行環境は以下のとおりである。

- OS MacOSX 10.7
- CPU 1.6GHz Intel Core 2 Duo
- メモリ 2GB 667MHz DDR2 SDRAM
- コンパイラ JDK1.6.0

JastAddJを用いた旧来の方法と、Javaバイトコードから構築する新しい方法、それぞれでJava ASTを構築し、構築にかかった時間を測定した。変換の対象としたのは、以下の４つのサンプルコードである。

- Sample0：制御構造のないコード
- Sample1：Whileが多重入れ子になっているコード
- Sample2：if-else文が複数回続くコード
- Sample3：While,if-else文が混合したコード

4.1.1　結果

実験をおこなう中で、１回のプログラム実行において、１回目のJava ASTの構築と２回目以降のJava ASTの構築では、実行時間に大きな差があることがわかった。そのため、１回目と２回目以降を分け、実行時間を計測した。１回目を計測した結果が図4.1、２回目以降を計測した結果が図4.2である。１回目のJava AST構築では、すべてのSampleにおいて、新しい手法の方が旧来の手法より実行時間が短いという結果が得ら

れた。また、２回目以降の Java AST 構築では、Sample 1 以外において、新しい手法の方が旧来の手法より実行時間が短いという結果が得られた。

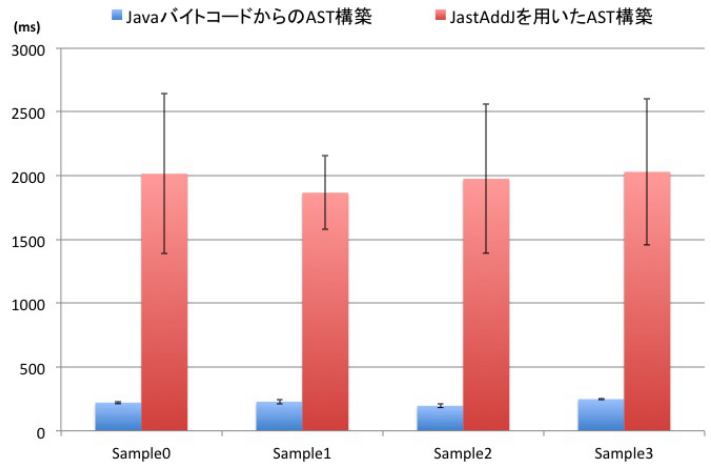

図 4.1: Java AST の構築にかかる時間 (1 回目)

4.1.2　考察

まず、１回目の Java AST の構築にかかる時間について考える。すべての Sample に対し、新しい手法の方が、Java AST の構築にかかる時間が短いという結果が得られた。旧来の手法では、JastAddJ を用いて Java ソースコードから Java AST を構築している。Java ソースコードの解析をおこなうためには、字句解析器や構文解析器を初期化する必要がある。そのため、旧来の手法では、１回目の Java AST 構築に長い時間を要したのではないかと考えられる。

次に、２回目以降の Java AST の構築にかかる時間について考える。Sample 1 以外、新しい手法の方が実行時間が短いという結果が得られた。Sample 1 は while の入れ子を含むようなコードである。新しい手法では、Java バイトコードから Java AST を構築している。構築をおこなう際に、制御構造の復元をおこなう必要があり、コントロールフローグラフと支配木を用いて解析をおこなっている。そのため、制御構造が入れ子になっている Sample 1 では、解析に長い時間を要したと考えられる。

２回目以降の Java AST の構築にかかる時間では、Sample1 において、新しい手法の方が時間がかかるという結果が出た。これから、制御構造が

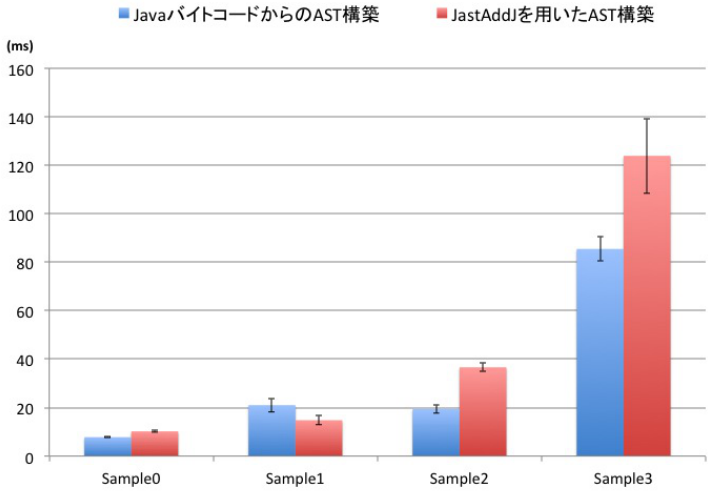

図 4.2: Java AST の構築にかかる時間 (2 回目以降)

多重入れ子になるようなものだと新しい手法は長い時間を必要とする可能性がある。しかし、システムの初期化にかかる時間は旧来の手法の方が大きくそのオーダーは 1000ms 単位である。それに対し、Java AST 構築自体にかかっている時間は 10ms 単位であり、初期化にかかる時間と比較すると小さい。したがって、1 回目、2 回目を総合して判断すると、Java AST の構築方法を新しい手法に変更したことによって、Java AST の構築にかかる時間を短縮することができたといえる。

4.2　Java と C 言語の配列アクセスの速度比較

巨大な配列を利用する場合、Java よりも C 言語の方が高速なことを検証するための実験をおこなった。また、この実験は、@type_alias、@fnc_alias の動作確認も兼ねている。実行環境は以下のとおりである。

- マシン Tsubame2.0 [14] Thin ノード
- OS Linux SUSE Linux Enterprise Server 11 SP1
- CPU Intel Xeon 2.93 GHz
- メモリ 54GB
- コンパイラ IBM J9 VM (build 2.4, JRE 1.6.0)

3 章で紹介した@type_alias、@fnc_alias を利用した BigArrayL クラス A.1 と、Java で実装した BigArrayL クラス A.2、それぞれを利用したプログラムを作成した。プログラムの内容は、配列の初期化、二分探索である。具体的には、図 4.3 に示した init メソッドと search メソッドを実行した。このうち、init メソッドの実行にかかった時間を測定した。ただし、実験の目的は C コードと Java コードの比較であるため、メソッドを呼び出す時間は含めていない。

```
1     void init(BigArrayL bal, long length) {
2       long i = 0;
3       while (i < length) {
4         bal.set(i, i);
5         i++;
6       }
7     }
8
9     long search(BigArrayL bal, long length, long value){
10      long f = 0;
11      long l = length - 1;
12      while(true){
13        if(f > l)
14          return -1;
15        long c = (f + l)/2;
16        if(bal.get(c) == value)
17          return c;
18        if(bal.get(c) < value)
19          f  = c + 1;
20        else
21          l = c - 1;
22      }
23    }
```

図 4.3: 二分探索をおこなうプログラム

4.2.1　結果

配列の初期化、二分探索とも正しく動作することを確認することができた。また、それぞれの init メソッドの実行にかかった時間は図 4.1 のようになった。図 4.1 において、WootinJ が@type_alias、@fnc_alias を利用したプログラム、Java が Java で実装したプログラムをそれぞれ表している。

表 4.1: init メソッドの実行時間 (ms)

	実行時間	標準偏差
WootinJ	6.241	0.050
Java	16.89	0.003

4.2.2　考察

実験結果から、WootinJ を利用し、@type_alias、@fnc_alias を使って実装したプログラムの方が実行時間が短いことがわかる。したがって、C 言語の方が Java よりも巨大配列を利用する場合、効率が良いといえる。今回の実験では、WootinJ のオーバーヘッドは計測していない。実際には WootinJ を用いた場合、WootinJ が C コードに変換、呼び出すためのオーバーヘッドがかかる。しかし、HPC のプログラムは実行時間が長く、WootinJ のオーバーヘッドが占める割合は小さいと考えられる。したがって、巨大な配列を用いた HPC 向けのプログラムを作成する場合、Java で実装するよりも WootinJ を用いた方が高速だと考えられる。

第5章　まとめと今後の課題

5.1　まとめ

本研究では、HPC 向け実行時言語変換器 WootinJ に対する 2 つの改善をおこなった。2 つの改善とは、Java AST 構築にかかる時間を短縮したことと表現力を拡張したことである。

WootinJ では、JastAddJ を用いて Java ソースコードから Java AST を構築していた。そのため、Java AST の構築に長い時間をかけていた。しかし、WootinJ は実行時にコードの変換をおこなうため、これは問題であった。そこで、Java バイトコードから Java AST を構築する新しい手法を考え、実装した。

WootinJ は HPC 向けの変換器であるが、HPC で有用なデータ構造を利用しづらいという問題があった。HPC で有用なデータ構造とは、長さが long で表現された配列や unsigned long である。これらを WootinJ で利用できるようにするために特殊なアノテーションを導入した。特殊なアノテーションとは、@type_alias と@fnc_alias である。これらのアノテーションを利用することで、型、メソッド呼び出しの変換を明示的に指定することができ、結果として先に挙げたようなデータ構造を用いることができるようになる。

実験を通して、本研究でおこなった 2 つの改善が意味のあるものであると確認することができた。まず、Java AST の構築にかかる時間を旧来の方法と新しい方法とで比較した。その結果、新しい方法の方が短い時間で Java AST を構築することができるという結論が得られた。また、@type_alias、@fnc_alias を用いて長さが long で表現されたクラス BigArrayL を作成し、それを用いて二分探索をおこなうプログラムを作成し、そのプログラムを Tsubame2.0 上で動作させ、期待どおりの結果を得ることができた。

5.2　今後の課題

現在の WootinJ には、まだ解決すべき問題が残っている。以下で、その問題を確認する。

5.2.1　WootinJ の複数ノードへの対応

現在の WootinJ は、1 ノードでの実行を前提としており、複数ノードに対する対応はできていない。@fnc_alias を利用することで、MPI などへの対応もできるのではないかと考えている。

5.2.2　@type_alias の改善

現在の@type_alias の引数には、任意の文字列を与えることができてしまう。しかし、@type_alias の引数として予想されるのは、C 言語における型である。そこで、@type_alias の引数として与えられた文字列が、C 言語の型として認識できるものか判定できるようにするべきだと考えている。

5.2.3　@fnc_alias の改善

現在の@fnc_alias はメソッドのレシーバが this であった場合に対応できていない。レシーバが this であった場合には、引数として this に相当するオブジェクトを要求することで解決できると考えている。

参考文献

[1] AMD: Aparapi, http://developer.amd.com/zones/java/aparapi/Pages/default.aspx.

[2] Ekman, T. and Hedin, G.: The jastadd extensible java compiler, *ACM Sigplan Notices*, Vol. 42, No. 10, ACM, pp. 1–18 (2007).

[3] Ganegoda, G., Samaranayake, D., Bandara, L. and Wimalawarne, K.: JConqurr-A Multi-Core Programming Toolkit for Java.

[4] Groovy: Groovy, http://groovy.codehaus.org/.

[5] JRuby: JRuby, http://jruby.org/.

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.: An overview of AspectJ, ECOOP 2001—Object-Oriented Programming, pp. 327–354 (2001).

[7] MPI: Message Passing Interface, http://www.mcs.anl.gov/research/projects/mpi/.

[8] NVIDIA: CUDA, http://developer.nvidia.com/cuda-gpus.

[9] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E. and Zenger, M.: An overview of the Scala programming language, Technical report, Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004).

[10] Proebsting, T., Townsend, G., Bridges, P., Hartman, J., Newsham, T. and Watterson, S.: Toba: Java for applications a way ahead of time (wat) compiler, *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)-Volume 3*, USENIX Association, pp. 3–3 (1997).

[11] Proebsting, T. and Watterson, S.: Krakatoa: Decompilation in Java (Does bytecode reveal source?), *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)* (1997).

[12] Ramshaw, L.: Eliminating go to's while preserving program structure, *Journal of the ACM (JACM)*, Vol. 35, No. 4, pp. 893–920 (1988).

[13] Tarjan, R.: Fast algorithms for solving path problems, *Journal of the ACM (JACM)*, Vol. 28, No. 3, pp. 594–614 (1981).

[14] Technology, T.: Tsubame computing services, `http://tsubame.gsic.titech.ac.jp/`.

[15] 冬彦丸山, 宏高小川, 聡松岡: 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, 情報処理学会論文誌プログラミング（PRO）, Vol. 40, No. SIG10(PRO5), pp. 39–50 (1999).

[16] 千葉滋: Javassist Home Page, `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

付 録A　プログラム例

```
1  @type_alias("long*")
2  public class BigArrayL {
3    private BigArrayL() {
4
5    }
6
7    public static BigArrayL factory(long length){
8      BigArrayL bal = new BigArrayL();
9      CRunner.register(bal, "make", length);
10     return bal;
11   }
12
13   static BigArrayL make(long length){
14     BigArrayL bal = new BigArrayL(length);
15     return bal;
16   }
17
18   @fnc_alias("(long *)malloc(@1 * sizeof(long))")
19   private BigArrayL(long length){
20
21   }
22
23   @fnc_alias("@0[@1]")
24   public long get(long index) {
25     return 0;
26   }
27
28   @fnc_alias("@0[@1] = @2")
29   public void set(long index, long value) {
30
31   }
32 }
```

図 A.1: @type_alias、@fnc_alias を用いた要素、長さが long で表現された配列の実装例

```
1  public class BigArrayL {
2    ArrayList<long[]> data = new ArrayList<long[]>();
3    int size = 1 << 20;
4
5    public BigArrayL(long length) {
6      int count = (int)(length/size);
7      for(long i = 0;i < count;i++){
8        long[] arr = new long[size];
9        data.add(arr);
10     }
11     long[] arr = new long[(int)(length % size)];
12     data.add(arr);
13   }
14
15   public void set(long index, long value) {
16     int count = (int)(index/size);
17     long[] arr = data.get(count);
18     int idx = (int)(index % size);
19     arr[idx] = value;
20   }
21
22   public long get(long index) {
23     int count = (int)(index/size);
24     long[] arr = data.get(count);
25     int idx = (int)(index % size);
26     return arr[idx];
27   }
28 }
```

図 A.2: Java の配列を複数個用いた長さが long で表現された配列の実装例