Javaへの変換による安全なC言語の実装
(Extended Abstract)

上嶋 祐紀　　住井 英二郎
東北大学 大学院 情報科学研究科

概 要

我々は，C 言語のプログラムをメモリ安全な Java 言語のプログラムに変換するトランスレータを実装した．そのような変換のためには，C 言語独特の操作であるポインタ演算を，Java プログラムで安全に模倣する必要がある．これを実現するために，我々は C 言語のポインタやメモリブロックを表現する Java のクラスを定義した．また，C 言語ではポインタと整数を相互にキャストすることが可能なので，整数もポインタと同様のオブジェクトに変換しなければならない．しかし，すべての整数をポインタと同様に表現すると大幅に効率が悪化する．そこで，データフロー解析により，ポインタが代入されない基本型の変数は，Java の通常の基本型変数に変換する，という最適化を実装した．9 個のベンチマークプログラムで実験したところ，最適化前の変換結果コードは元の C プログラムに対し 5.5 倍〜87 倍程度の実行時間がかかったが，最適化後は（元の C プログラムに対し）1.6 倍〜39 倍程度に改善した．

1　序論

一般にプログラミング言語のメモリ安全性は，バグや攻撃からプログラムやデータを保護するために有用な性質である．しかし，現在，もっとも広く使われているプログラミング言語の一つである C 言語では，メモリ安全性が保証されておらず，予期せぬ動作やセキュリティホールの原因となっている．

そこで我々は Java 言語がメモリ安全とされていることを利用し，C 言語のプログラムを Java へ変換することにより，C プログラムのメモリ安全性を向上する手法を提案する．我々の方針は，

- C 言語の仕様で定義されている動作は，変換後の Java プログラムにおいても模倣しつつ，
- 「未定義」とされている動作は，実行時検査により確実にエラーとするか，あるいは安全な（すなわち未定義でない）動作で置き換える，

というものである [8]．C のプログラムを Java へ変換することにより，Java 処理系のメモリ安全性を仮定すれば，C プログラムのメモリ安全性も保証することができる．また，プログラムを Java バイトコードで配布したり，ブラウザ上でアプレットとして動かすといったことも可能になる．

C 言語と Java 言語のもっとも大きな違いは，メモリモデルとポインタ演算である．我々はまず C のポインタ演算を Java で模倣するために，ポインタ演算を安全に実装するためのデータ構造である Fat Pointer [1, 8] に着目した．Fat Pointer はポインタを通常の 1 ワードから 2 ワードに拡張したもので，1 ワード目は常にメモリブロックの先頭を指す「ベース」，2 ワード目は現在指しているメモリがベースから何バイト離れているかを示す「オフセット」である．我々は，これを Java の FatPtr クラスとして実装した．

我々はまた，C 言語の「メモリブロック」（連続したメモリ領域）を Java の配列により模倣する Block クラスを実装した．我々の実装は,「メモリブロックをどのように読み書きするべきか」を表すアクセスメソッド [7] を各々のメモリブロック自身に持たせることにより，型の異なるポインタ間のキャストもサポートしている．

以上の Fat Pointer，メモリブロック，アクセスメソッドといった概念は，安全な C 言語処理系である Safe C [1] および Fail-Safe C [7, 8] に由来するものである．本研究は，コンパイラの生成する出力を低水準

```
abstract class Fat {
    Block base;
    int offset;
    abstract int asInt();
}
```

```
class FatPtr extends Fat {
    FatPtr(Block b, int n) {
        base = b;
        offset = n;
    }
    int asInt() {
        if(this.base == null)
            return this.offset;
        else
            return this.base.addr + this.offset;
    }
}
```

図 1: Fat 抽象クラスと FatPtr クラス

なコードではなく,(元からオブジェクトやメソッドなどの仕組みを持つ) 高水準な Java プログラムとしたことにより，これらの概念をより明確かつ単純に実装している．

2 変換に利用する Java のクラス

1 節で述べたように，我々は C 言語のポインタとメモリブロックを，Java の FatPtr クラス（図 1）および Block クラス（図 2）として実装する．

FatPtr クラスは，Block クラス型のフィールド base と，int 型のフィールド offset を持つ．また，ポインタを整数として表示したときの値を返すメソッド asInt を持つ．asInt は，base の指す Block オブジェクトの addr フィールド（後述）の値と，offset との和を返す．ただし，base が null のときは offset の値を返す．

また，C 言語では整数とポインタを相互にキャストすることが可能なので，一般には整数もポインタと同様に表現しなければならない [8]．我々はこれを実現するために，C 言語の整数を表現する Java のクラス FatInt を実装し，FatPtr と FatInt に共通する親クラスとして Fat 抽象クラス（図 1）を作成した．Fat 抽象クラスは，FatPtr と同じ型のフィールドおよびメソッドを持つ．FatInt クラスは，base が常に null であるような，Fat クラスの子クラスである[1]．

Block クラスは実際には抽象クラスであり，格納される要素の型ごとに ByteBlock クラス，DoubleBlock クラス，FatBlock クラス（図 2）などの子クラスを持つ．これらのクラスは，連続したメモリ領域を表現する配列である contents フィールドと，整数型のフィールド size，addr，objsize を持つ．size フィールドは，メモリブロックのサイズを表す．addr フィールドは，メモリブロックの先頭アドレスを表す整数を持つ．この整数は，Block オブジェクトが生成されるたびに,「現在のヒープポインタ」を模倣するグローバル変数 AddrCounter から計算される．objsize フィールドは，メモリブロックに格納されている要素の（C 言語における）サイズを表す．

さらに，Block クラス（およびその子クラス）は，各型の値を読み書きするためのアクセスメソッド readFat，readByte，readDouble，writeFat，writeByte，writeDouble 等々を持つ．一般に型 X に対し，メソッド readX はオフセットを表す整数 vo を受け取り，contents から X 型の値を読み出して返す．また，メソッド writeX は整数 vo と X 型の値 x を受け取り，contents に x を書き込む．ただし，vo がメモリブロックを逸脱するなど安全でない（C 言語の仕様で動作が未定義とされている）アクセスについては例外を発

[1] したがって，FatInt を単に FatPtr の子クラスとする実装も可能である．しかし，我々の実験では，Fat 抽象クラスを利用した場合のほうが 0%〜20%程度高速であった．これは HotSpot VM による最適化の影響と思われるが，まだ具体的には確認できていない．

```
abstract class Block {
    int objsize;
    int size;
    int addr;
    Object[] contents;
    abstract Fat readFat(int vo);
    abstract void writeFat(int vo, Fat f);
    abstract Byte readByte(int vo);
    abstract void writeByte(int vo, Byte b);
    ...
}
```

```
class FatBlock extends Block {
    Fat[] contents;
    FatBlock(int n) {
        objsize = 4;
        size = 4 * n;
        addr = AddrCounter.getAddr();
        AddrCounter.incrAddr(size);
        contents = new Fat[n];
    }
    Fat readFat(int vo) {
        if(vo % 4 == 0)
            return this.contents[vo / 4];
        else
            // 例外を発生
    }
    ...
}
```

図 2: Block 抽象クラスと FatBlock クラス

生する．Block クラスはあくまで C 言語のメモリ領域を Java の配列で模倣しているだけなので，そのような検査は容易である．

3　基本的な変換法

2 節で述べたクラスを利用して，我々は本節で述べるように変換を実現した．

まず，C 言語の配列は Block クラス（の子クラス）のオブジェクトに変換する．たとえば char の配列は ByteBlock オブジェクトに変換する．ポインタおよび int の配列は，型に関わらず，すべて FatBlock とする．配列の読み書きは，Block オブジェクトに対するアクセスメソッドの呼び出しとする．配列の要素へのポインタ`&a[i]`は，FatPtr の base に`a`（への参照）を，offset に`i * a.objsize`を代入することで容易に表現できる．

配列以外の変数は，1 要素の配列とみなす．すなわち，変数宣言`X x`は`XBlock x = new XBlock(1)`，変数参照`x`は`x.readX(0)`，変数への代入`x = ...`は`x.writeX(0, ...)`のように変換する．これにより，変数へのポインタ`&x`は，base が Block オブジェクトへの参照`x`で，offset が`0`の FatPtr とすることができる．

ポインタ演算は，FatPtr の offset を使用することで自然に実現できる．（C 言語の仕様では，連続しないメモリブロックへのポインタ同士の引き算は結果が未定義とされているので，base が異なる FatPtr 間の引き算を実現する必要はない．）

式$_1$ `*` 式$_2$ のような整数演算は，式$_1$ の変換結果を E_1，式$_2$ の変換結果を E_2 として，`new FatInt(`E_1`.asInt() * `E_2`.asInt())` のように変換することができる．

キャストは次の通りである．ポインタ間のキャストは，何もしなくて良い．これは（ポインタではなく）メモリブロック自身がアクセスメソッドを保持しているためである．また，整数とポインタ間のキャストも何もしなくて良い．これは，どちらも Fat クラス（の子クラス）のオブジェクトで表現されているためである．

付録 A に一般的な変換規則と変換例を掲載する．

4　トランスレータの実装

我々は，3 節で述べた基本的な変換法（および 5 節の最適化）に基づき，C プログラムを入力として Java プログラムを出力するトランスレータを Objective Caml で実装した．C プログラムの字句・構文解析には，CIL ライブラリ [5] を利用した．また，Java プログラムの pretty printer としては，Joust ライブラリ [2] を利用した．

5　最適化

3 節で述べた基本的な変換法では，ポインタとまったく関係のない通常の整数も，すべて FatInt オブジェクトに変換されてしまう．さらに，C の基本型の変数も，すべて Java の Block クラスのオブジェクトに変換され，読み書きのたびにアクセスメソッドが呼び出される．そのため，オブジェクト生成やメソッド呼び出しが大量に起こり，通常の C プログラムの実行と比べ，大きなオーバーヘッドとなる．

これらのオーバーヘッドを削減するために，我々はデータフロー解析により以下の条件をすべて満たす変数を求め，それらを（Block オブジェクトではなく）Java の通常の基本型の変数に変換するように，4 節の実装を改良した．

- int, long, double といった，C の基本型を持つ．
- ポインタを代入されることがない．
- アドレス演算子 (&) によりアドレスを取られることがない．
- 関数の仮引数ではない．

さらに，C の基本型の値は Java の（オブジェクトではなく）基本型の値に変換し，必要になるまでオブジェクトを生成しないようにした．ただし「必要になる」とは以下のいずれかの場合である．

- 配列，構造体，ないし Block オブジェクトに変換された変数に代入される．
- 関数の実引数または返り値になる．

これらの最適化により，6 節で示すように，プログラム実行時間のある程度の削減に成功した．

6　実験結果と考察

3 節で述べた変換法および 5 節で述べた最適化を評価するために，The Computer Language Shootout Benckmarks[2] より 9 個の C プログラム（表 1）を用いて実験を行った．実験に用いた PC の CPU は Pentium 4 2.80 GHz，メインメモリは 2 GB である．OS は Linux 2.6，Java コンパイラおよび Java 仮想マシンには JDK 1.5.0，比較のための C コンパイラとして GCC 4.0.0 に-O3 オプションをつけて用いた．

表 2 において unoptimized の行の値は，最適化を施していない Java プログラムの HotSpot VM での実行時間と，元の C プログラムを GCC でコンパイルして実行したときの時間との比である．optimized の行の値は，最適化を施した Java プログラムの実行時間と，元の C プログラムの実行時間との比である．

また，hand-written の行の値は，（2 節の Java クラスや 4 節のトランスレータを使わずに）手書きした Java プログラムの実行時間と，元の C プログラムの実行時間との比である．これにより，C 処理系に対する Java 処理系そのもののオーバーヘッドを見積もった．この値と，optimized の値が一致することが理想となる．

[2] http://shootout.alioth.debian.org

nsieve	素数の個数を計算する．ループ内で配列を用いて整数演算を行う．
spectral-norm	正方行列の 2 ノルムを計算する．浮動小数の配列で表された行列・ベクトルの演算を行う．
mandelbrot	マンデルブロ集合を計算する．ループ内で浮動小数演算を行う．
recursive	アッカーマン関数の値，フィボナッチ数，およびたらいまわし関数の値を計算する．再帰関数内で整数演算あるいは浮動小数演算を行う．
partial-sums	様々な数列の部分和を計算する．ループ内で浮動小数演算を行う．
nsieve-bits	素数の個数を計算する．ループ内で配列を用いてビット演算を行う．
fannkuch	順列の並べ換えを行う．ループ内で配列を用いて整数演算を行う．
binary-trees	二分木の生成と解放を繰り返す．ループ内で構造体の確保，読み書き，解放を行う．
n-body	惑星運動のシミュレーションを行う．ループ内で構造体の配列を用いて浮動小数演算を行う．

表 1: ベンチマークプログラムの概要

	nsieve	spectral-norm	mandelbrot	recursive
unoptimized	49.91	57.22	44.84	65.85
optimized	27.52	26.81	3.90	23.29
hand-written	3.06	1.27	3.55	1.87

	partial-sums	nsieve-bits	fannkuch	binary-trees	n-body
unoptimized	5.43	87.31	50.87	42.62	69.67
optimized	1.63	37.94	10.72	38.99	38.68
hand-written	1.65	17.84	2.34	1.35	1.95

表 2: 性能評価実験の結果（GCC による実行時間を 1 とする）

表 2 において，mandelbrot と partial-sums では，optimized と hand-written の値にほとんど差がない．これは，5 節の最適化により，手書きの場合とほぼ同じ Java プログラムをトランスレータが生成したためである．一方，それ以外のプログラムでは，最適化の効果はあるものの，理想 (hand-written) との差が大きい．C のソースプログラムと，変換された Java コードを観察すると，原因は以下の通りであると考えられる．

1. binary-trees と n-body は全プログラムにわたって構造体を用いており，構造体のメンバがすべて Block クラスのオブジェクトとなることによるオーバーヘッドが大きい．

2. さらに，binary-trees と n-body はループや再帰関数の中でポインタを用いており，FatPtr クラスのオーバーヘッドがある．

3. nsieve, spectral-norm, recursive, nsieve-bits, fannkuch, binary-trees, n-body では，関数の引数が Block クラスのオブジェクトとなり，ループや再帰関数の中でアクセスされることによるオーバーヘッドが大きい．

4. また，nsieve, spectral-norm, nsieve-bits, fannkuch, n-body では，配列が Block クラスのオブジェクトとなることによるオーバーヘッドがある．

1 番目の問題は，現在の構造体の実装の不備が原因であり，構造体のメンバを Block としない実装も原理的には容易である．また，3 番目の問題については，関数定義 f(t x) { ... } を f(t x') { t x = x'; ... } のように書き換えるだけで，x' は（決してアドレスを取られないので）Block とする必要がなくなる．さらに，5 節の条件を満たせば，x を最適化することも可能である．一方，2 番目と 4 番目の問題を解決するためには

- オフセットが常に 0 であるようなポインタを探し出し，通常の Java の参照に変換する
- アクセスメソッドを必要としない配列を検出し，通常の Java の配列に変換する

など,（典型的ではあるが）別個の解析・最適化が必要となる．

7 関連研究

本研究以外にも，C から Java へのトランスレータはいくつか存在する [3, 4]．これらのトランスレータは，C プログラムを Java 環境で安全に実行するというより，C プログラムを Java 言語に移植するための支援ツールであり，ポインタのキャストなど，Java 言語に存在しない機能はあまりサポートしていない．これに対し，我々のトランスレータは，たとえばアクセスメソッドを使用することにより，キャストされたポインタによるメモリアクセスにも対応している．

CCured [6] は，静的解析と実行時検査により，C プログラムを安全かつ高速に実行するためのコンパイラである．CCured は C から C へのトランスレータとして実装されており，出力されるコードの安全性は，コンパイラの正当性によってのみ保証されている．また，ポインタやキャストの実現が十分ではなく，ポインタと整数のキャストに対応していない．

Fail-Safe C [7, 8] は，C プログラムを安全に実行するためのコンパイラである．Fat Integer やアクセスメソッドの概念は，この研究に由来する．CCured と同様に，Fail-Safe C も C から C へのトランスレータとして実装されている．我々の研究は，コンパイラの出力を低水準なアセンブリないし C コードではなく高水準な Java プログラムとしたことにより，Java の安全性やクラス機構を利用して，Fail-Safe C の概念をより明確かつ単純に実現している．

8　結論と今後の課題

本論文では，Cプログラムを Java へ変換することにより，Cプログラムのメモリ安全性を確保する手法を提案した．

本研究の今後の課題の一つに，ANSI C のフルセットをサポートすることがある．現在，我々がサポートしていない ANSI C 言語仕様のうち，実現方法が自明でないもの（および我々が想定している実現方法）は次の通りである．

符号なし整数 Java には C の符号なし (unsigned) 整数に相当するデータ型がないので，符号つき整数や，より大きな整数型を用いて間接的にサポートすることを考えている．

共用体 共用体は，それぞれのメンバに対応する Java のクラスを定義し，それらに共通の抽象親クラスを定義すれば実現できると考えている．

関数ポインタ 関数ポインタは，その関数に相当するメソッドを持つオブジェクトへの参照で実現できると思われる．

多次元配列 多次元配列は Java にもあるが，C では多次元配列は連続したメモリ領域に確保されることが要求されており，アクセスメソッドで対応する必要がある．

goto 文 C の goto 文は，Java のラベル付き break 文およびラベル付き continue 文に変換可能であると思われる．

これに関連して，標準ないし既存の C ライブラリへの対応も課題である．現在は

- ライブラリをソースコードからコンパイルし直す
- ライブラリのためのラッパーを書く
- ライブラリを Java で書き直す

という 3 つのアプローチを併用することを想定している．また，Cプログラム（を変換した Java プログラム）から Java のライブラリを利用する方法についても検討が必要である．

以上のような方針により，実際に既存の Cプログラムをどこまでサポートできるかは，確認が必要である．当座の目標として，SPEC CPU ベンチマーク（の大半）を動作させることを目指したいと考えている．

6 節の終わりで述べたような最適化も今後の課題である．また，変換された Java コードについて，JIT コンパイラがどのような最適化をどこまで自動的に行うか（あるいは行わないか），できるだけ具体的に調べたいと考えている．

本研究では，C プログラムを変換する対象言語として，安全な言語の代表である Java を選択したが，C#や ML など，Java 以外の安全な言語への変換も可能であると考えている．その場合，変換されたプログラムの性能や，C 言語の仕様をどこまでサポートできるかについては研究が必要である．

謝辞

本研究にあたり，東北大学 小林・住井研究室の皆様と産業技術総合研究所 情報セキュリティ研究センターの大岩寛氏から様々な助言とサポートをしていただきました．また，査読者の方々からも数々の有益な助言をいただきました．ここに感謝の意を表します．

参考文献

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Programming Language Design and Implementation*, pp. 290–301, 1994.

[2] Eric Cooper. Java parser and pretty-printer. `http://www.cs.cmu.edu/~ecc/software.html`.

[3] Jazillian: The legacy to "natural" Java translator. `http://www.jazillian.com`.

[4] Johannes Martin and Hausi Muller. Strategies for migration from C to Java. In *Software Maintenance and Reengineering*, pp. 200–209, 2001.

[5] George Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pp. 213–228, 2002.

[6] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages*, pp. 128–139, 2002.

[7] Yutaka Oiwa. *Implementation of a Fail-Safe ANSI C Compiler*. PhD thesis, University of Tokyo, 2004.

[8] 大岩寛, 住井英二郎, 米澤明憲. 安全性を保証する ANSI-C 実行系の実装手法. 日本ソフトウェア科学会第 18 回大会, 2001 年 9 月.

A 抽象構文と変換規則

本研究で扱う抽象構文（C のサブセット）は以下の通りである.

$$
\begin{array}{rcl}
\text{program} & ::= & \text{str program} \mid \text{function program} \mid \text{mainfunction} \\
\text{str} & ::= & \textbf{struct}\ \text{id}\{\text{strdecl}^*\} \\
\text{function} & ::= & \text{typ id}(\text{argument}^*)\{\text{decl}^*\ \text{stmt}^*\} \\
\text{mainfunction} & ::= & \textbf{void}\ \text{main}()\{\text{decl}^*\ \text{stmt}^*\} \\
\text{strdecl} & ::= & \text{typ id} \mid \text{typ id[exp]} \\
\text{argument} & ::= & \text{typ id} \mid \text{typ id[]} \\
\text{decl} & ::= & \text{typ id} \mid \text{typ id[exp]} \\
\text{typ} & ::= & \textbf{void} \mid \textbf{int} \mid \textbf{char} \mid \textbf{short} \mid \textbf{long long} \mid \textbf{double} \mid \text{typ}* \mid \textbf{struct}\ \text{id} \\
\text{exp} & ::= & \text{constant} \mid \text{lval} \mid \text{unop exp} \mid \text{exp1 binop exp2} \mid (\text{typ})\text{exp} \mid \&\text{lval} \\
\text{unop} & ::= & - \mid\ \sim\ \mid ! \\
\text{binop} & ::= & + \mid - \mid * \mid / \mid \% \mid <<\ \mid >>\ \mid <\ \mid >\ \mid <= \mid >= \mid ==\ \mid != \\
 & \mid & \&\mid \wedge \mid\ \mid \\
\text{lval} & ::= & (\text{lhost}, \text{offs}) \\
\text{lhost} & ::= & \text{id} \mid *\text{exp} \\
\text{offs} & ::= & \text{nooffset} \mid \text{.id offs} \mid \text{[exp] offs} \\
\text{stmt} & ::= & \text{instr}^* \mid \textbf{return}(\text{exp}^?) \mid \textbf{break} \mid \textbf{continue}
\end{array}
$$

$$
\begin{array}{rcl}
 & | & \textbf{if}(\text{exp})\{\text{stmt1}^*\}\textbf{else}\{\text{stmt2}^*\} \\
 & | & \textbf{while}(1)\{\text{stmt}^+\} \\
\text{instr} & ::= & \text{lval} = \text{exp} \mid \text{lval}^? = \text{id}(\text{exp}^*)
\end{array}
$$

ただし，太字は C における予約語である．id は C において識別子として使える文字列である．肩の*（decl*など）は，それが 0 回以上連続して出現することを意味する．mainfunction の構文は，main 関数の返り値の型を void に，引数を 0 個に限定していることを示している．constant は定数である．nooffset は，offs がこれ以上続かないことを表す．添字の?（$\text{exp}^?$ など）は，それがある場合とない場合があることを意味する．while の構文中に現れる $^+$ は，stmt が 1 回以上連続して出現することを意味する．このうち一つは if 文であり，while 文はこの if 文を利用し，ループを抜けるかどうかを判断する．$\text{lval}^? = \text{id}(\text{exp}^*)$ において，id は関数名である．

for 文と do-while 文は，上の抽象構文には含まれていないが，CIL によって if 文と while 文の組合せに変換される．

前述の抽象構文に従った C プログラムを Java に変換する規則は以下の通りである．変換を表す記号として [[]] を用いる．

$$
\begin{array}{rcll}
[\![\text{str program}]\!] & \rightarrow & [\![\text{str}]\!]\ [\![\text{program}]\!] & (1) \\
[\![\text{function program}]\!] & \rightarrow & [\![\text{function}]\!]\ [\![\text{program}]\!] & (2) \\
[\![\ \textbf{void}\ \text{main}()\{\text{decl}^*\ \text{stmt}^*\}]\!] & \rightarrow & \textbf{void}\ \text{main}(\text{string}[]\ \text{args})\{[\![\text{decl}]\!]^*\ [\![\text{stmt}]\!]^*\} & (3) \\
 & & & \\
[\![\textbf{struct}\ \text{id}\{\text{strdecl}^*\}]\!] & \rightarrow & \textbf{class}\ \textit{id}\{[\![\text{strdecl}]\!]^*\} & \\
 & & \textbf{class}\ \textit{id}\text{Block}\ \textbf{extends}\ \text{Block}\{\ldots\} & (4) \\
[\![\text{typ id}(\text{argument}^*)\{\text{decl}^*\ \text{stmt}^*\}]\!] & \rightarrow & \textit{objtype}\ \text{id}([\![\text{argument}]\!]^*)\{[\![\text{decl}]\!]^*\ [\![\text{stmt}]\!]^*\} & (5) \\
 & & & \\
[\![\text{typ id}]\!] & \rightarrow & \textit{objtype}\ \text{id} & (6) \\
[\![\text{typ id[exp]}]\!] & \rightarrow & \textit{objtype}\ \text{id}[[\![\text{exp}]\!].\text{asInt}()] & (7) \\
[\![\text{typ id}]\!] & \rightarrow & [\![\text{typ}]\!]\ \text{id} & (8) \\
[\![\text{typ id}[]]\!] & \rightarrow & [\![\text{typ}]\!]\ \text{id} & (9) \\
[\![\text{typ id}]\!] & \rightarrow & [\![\text{typ}]\!]\ \text{id} = \textbf{new}\ [\![\text{typ}]\!](1) & (10) \\
[\![\text{typ id[exp]}]\!] & \rightarrow & [\![\text{typ}]\!]\ \text{id} = \textbf{new}\ [\![\text{typ}]\!]([\![\text{exp}]\!].\text{asInt}()) & (11) \\
[\![\textbf{void}]\!] & \rightarrow & \textbf{void} & (12) \\
[\![\textbf{int}]\!] & \rightarrow & \text{FatBlock} & (13) \\
[\![\textbf{char}]\!] & \rightarrow & \text{ByteBlock} & (14) \\
[\![\textbf{short}]\!] & \rightarrow & \text{ShortBlock} & (15) \\
[\![\textbf{long long}]\!] & \rightarrow & \text{LongBlock} & (16) \\
[\![\textbf{double}]\!] & \rightarrow & \text{DoubleBlock} & (17) \\
[\![\text{typ}*]\!] & \rightarrow & \text{FatBlock} & (18) \\
[\![\textbf{struct}\ \text{id}]\!] & \rightarrow & \textit{id}\text{Block} & (19) \\
 & & & \\
[\![\text{constant}]\!] & \rightarrow & \textbf{new}\ \textit{objtype}(\text{constant}) & (20) \\
[\![-\text{exp}]\!] & \rightarrow & \textbf{new}\ \textit{objtype}(-[\![\text{exp}]\!].\textit{toobj}()) & (21)
\end{array}
$$

$$
\begin{aligned}
[\![\sim \mathrm{exp}]\!] &\rightarrow \textbf{new}\ \mathit{objtype}(\sim [\![\mathrm{exp}]\!].\mathit{toobj}()) && (22)\\
[\![! \ \mathrm{exp}]\!] &\rightarrow \textbf{new}\ \mathrm{FatInt}([\![\mathrm{exp}]\!].\mathit{toobj}() == 0\ ?\ 1\ :\ 0) && (23)\\
[\![\mathrm{exp1}\ \mathit{arithop}\ \mathrm{exp2}]\!] &\rightarrow \textbf{new}\ \mathit{objtype}([\![\mathrm{exp1}]\!].\mathit{toobj}()\\
&\qquad \mathit{arithop}\ [\![\mathrm{exp2}]\!].\mathit{toobj}()) && (24)\\
[\![\mathrm{exp1}\ \mathit{ptrop}\ \mathrm{exp2}]\!] &\rightarrow \mathrm{Fat\ temp} = [\![\mathrm{exp1}]\!];\\
&\qquad \textbf{new}\ \mathrm{FatPtr(temp.base,\ temp.offset}\ \mathit{ptrop}\\
&\qquad [\![\mathrm{exp2}]\!].\mathit{toobj}() * \mathrm{temp.base.objsize}) && (25)\\
[\![\mathrm{exp1}\ \mathit{compop}\ \mathrm{exp2}]\!] &\rightarrow \textbf{new}\ \mathrm{FatInt}([\![\mathrm{exp1}]\!].\mathit{toobj}()\\
&\qquad \mathit{compop}\ [\![\mathrm{exp2}]\!].\mathit{toobj}()\ ?\ 1\ :\ 0) && (26)
\end{aligned}
$$

$$
\begin{aligned}
[\![\textbf{return}(\mathrm{exp}^{?})]\!] &\rightarrow \textbf{return}([\![\mathrm{exp}]\!]^{?}) && (27)\\
[\![\textbf{break}]\!] &\rightarrow \textbf{break} && (28)\\
[\![\textbf{continue}]\!] &\rightarrow \textbf{continue} && (29)\\
[\![\textbf{if}(\mathrm{exp})\{\mathrm{stmt1}^{*}\}\textbf{else}\{\mathrm{stmt2}^{*}\}]\!] &\rightarrow \textbf{if}([\![\mathrm{exp}]\!].\mathit{toobj}()\ !=\ 0)\\
&\qquad \{[\![\mathrm{stmt1}]\!]^{*}\}\textbf{else}\{[\![\mathrm{stmt2}]\!]^{*}\} && (30)\\
[\![\textbf{while}(1)\{\mathrm{stmt}^{+}\}]\!] &\rightarrow \textbf{while}(\mathrm{true})\{[\![\mathrm{stmt}]\!]^{+}\} && (31)
\end{aligned}
$$

ただし，矢印の右辺の太字は Java における予約語である．(4)，(19) 中の *id* は，矢印の左辺の id の先頭を大文字に変換した文字列である（Java では，クラス名の先頭を大文字とすることが推奨されている）．(5) の *objtype* は，左辺の typ が void，int，char，short，long long，double，typ∗，**struct** id であれば，それぞれ void，FatInt，Byte，Short，FatLong，Double，FatPtr，*id*（クラス型）となる．FatLong は C の long long に対応する Java のクラスである．(6)，(7) の *objtype* は，左辺の typ が int，char，short，long long，double，typ∗，**struct** id であれば，それぞれ FatInt，Byte，Short，FatLong，Double，FatPtr，*id*（クラス型）となる．また，(6)，(7) は strdecl の変換規則である．exp の変換規則（(20) など）に現れる *objtype* は，C において exp の型に対応する Java オブジェクトの型を表す．(8)，(9) は argument の変換規則であり，(10)，(11) は decl の変換規則である．(21) などに現れる *toobj* は，オブジェクトに適用すると，そのオブジェクトが保持する定数を返すメソッドを表す．オブジェクトの型が FatInt，Byte，Short，FatLong，Double，FatPtr であれば，*toobj* はそれぞれ asInt，byteValue，shortValue，asLong，doubleValue，asInt となる．(24) に現れる *arithop* は，+，−，∗，/，%，<<，>>，&，ˆ，| のいずれかである．(24) の 2 つの exp は基本型である．(25) に現れる *ptrop* は，+，− のいずれかである．(25) の exp1 はポインタ型であり，exp2 は int 型である．(26) に現れる *compop* は，<，>，<=，>=，==，!= のいずれかである．(26) の 2 つの exp は基本型である．

(4) は構造体宣言を Java に変換する規則であるが，（構造体を模倣する）*id* クラスとともに *id*Block クラス（Block 抽象クラスの子クラス）が生成されることを意味している[3]．(5)，(3) で生成される Java のメソッドは，元の C プログラムが書かれたファイルの名前の，先頭を大文字にしたクラスの中に定義される．(3) が示すように，C の main 関数は，引数が (String[] args) であるような Java の main メソッドに変換される[4]．(7)，(11) では asInt メソッドが用いられている．これはトランスレータが，C において配列の要素数の指定に int を用いるプログラムのみ，Java への変換をサポートしているからである．(10) が示すように，変数は 1 要素の配列を模倣するオブジェクトに変換される．(18) が示すように，いかなるポイン

[3] *id*Block クラスはアクセスメソッドを持たなければならないが，現在の実装では，構造体読み出し用のメソッド以外は備えていない．

[4] したがって現在の実装では，main 関数の引数を使用する C プログラムを Java に変換した場合，その Java プログラムは正常に動作しない．

タ型であっても全て FatBlock 型に変換される．(19) が示すように，構造体型は，構造体を模倣するオブジェクトを格納するメモリブロックの型に変換される．(20)〜(26) が示すように，exp の変換結果は必ずオブジェクトとなる．(23) の右辺は，値 0 か 1 のどちらかを持つ FatInt オブジェクトを生成することを意味する．(25) の右辺は，offset の値を（C における exp2 の評価結果 * ポインタが参照するメモリブロックの objsize）だけ増減させた新しい FatPtr オブジェクトを作成することを意味する．(26) の右辺は，Java における比較演算の結果が true なら値 1，false なら値 0 を持つ FatInt オブジェクトを作成することを意味する（Java における比較演算の結果は boolean 型の値であることを考慮している）．(30) が示すように，if 文の条件式は，Java において評価したときの返り値が 0 かどうかを判定するようなコードに変換される．すなわち，生成される Java コードは，（条件式の返り値が 0 なら偽とし，それ以外なら真とする）C の条件判定を模倣する．(31) が示すように，while 文の条件式である 1 は true に変換される．

以下で，我々はその他の変換規則について個別に説明する．

exp1 − exp2　変換規則は以下の通りである．

$$
\begin{aligned}
[\![\mathrm{exp1} - \mathrm{exp2}]\!] \quad \rightarrow \quad & \mathrm{Fat\ temp1} = [\![\mathrm{exp1}]\!]; \\
& \mathrm{Fat\ temp2} = [\![\mathrm{exp2}]\!]; \\
& \textbf{if}(\mathrm{temp1.base.addr} != \mathrm{temp2.base.addr})\{ \\
& \quad //\text{例外を発生} \\
& \} \\
& \textbf{new}\ \mathrm{FatInt}((\mathrm{temp1.offset} - \mathrm{temp2.offset}) / \mathrm{temp1.base.objsize}) \qquad (32)
\end{aligned}
$$

ただし，(32) は，2 つの exp がポインタ型の場合のみ適用される．(32) の右辺の Java プログラムは，2 つの FatPtr オブジェクトの base が参照するメモリブロックの addr の値を比較し，それが等しくない場合は（C 言語の仕様では，連続しないメモリブロックへのポインタ同士の引き算は結果が未定義とされているので）例外を発生する．等しい場合は，2 つの FatPtr オブジェクトが参照するメモリが互いに何バイト離れているかを表す値を，FatPtr オブジェクトの base が参照するメモリブロックの objsize の値で除算する．

typ(exp)　変換規則は以下の通りである．

$$
\begin{aligned}
[\![(\mathrm{typ})\mathrm{exp}]\!] &\rightarrow \textbf{new}\ \mathit{objtype}((\mathit{casttype})[\![\mathrm{exp}]\!].\mathit{toobj}()) && (33)\\
[\![(\mathrm{typ}*)\mathrm{exp}]\!] &\rightarrow [\![\mathrm{exp}]\!] && (34)\\
[\![(\textbf{int})\mathrm{exp}]\!] &\rightarrow [\![\mathrm{exp}]\!] && (35)\\
[\![(\textbf{long long})\mathrm{exp}]\!] &\rightarrow [\![\mathrm{exp}]\!].\mathrm{toLong}() && (36)\\
[\![(\mathrm{typ}*)\mathrm{exp}]\!] &\rightarrow [\![\mathrm{exp}]\!] && (37)\\
[\![(\mathrm{typ}*)\mathrm{exp}]\!] &\rightarrow [\![\mathrm{exp}]\!].\mathrm{toFat}() && (38)\\
[\![(\mathrm{typ}*)\mathrm{exp}]\!] &\rightarrow \textbf{new}\ \mathrm{FatInt}((\textbf{int})[\![\mathrm{exp}]\!].\mathit{toobj}()) && (39)
\end{aligned}
$$

ただし，(33) は，typ が基本型でかつ C における exp が基本型の場合のみ適用される．*casttype* は，typ が int，char，short，long long，double であれば，それぞれ int，byte，short，long，double となる．(34)，(35)，(36) は，exp がポインタ型の場合のみ適用される．(36) の toLong は，FatPtr オブジェクトに適用すると FatLong オブジェクトを返すメソッドである．(37) は，exp が int 型の場合のみ適用される．(38) は，exp が long long 型の場合のみ適用される．toFat は，FatLong オブジェクトに適用すると FatPtr オブジェクトを返すメソッドである．(39) は，exp が char 型あるいは short 型の場合のみ適用される．

(33) が示すように，基本型を他の基本型にキャストする C プログラムは，Java のキャストを用いる Java プログラムに変換される．(34) はポインタから型の異なるポインタへのキャストを，(35) はポインタから

int へのキャストを表す規則である．これらの規則で生成される Java プログラムは，式の評価以外何も行わない．(36) はポインタから long long へのキャストを表す規則である．(37) は int からポインタへのキャストを表す規則である．この規則で生成される Java プログラムは，式の評価以外何も行わない．(38) は long long からポインタへのキャストを表す規則である．(39) は char あるいは short からポインタへのキャストを表す規則である．C において，char や short をポインタにキャストしてもポインタとして使用できないことから，(39) の右辺は FatInt オブジェクトを作成するプログラムとなっている．

lval　変換規則は以下の通りである．

$$\begin{aligned}
[\![(\mathrm{id}, \mathrm{offs})]\!] &\rightarrow \mathrm{id}.\mathit{read}([\![\mathrm{offs}]\!]_t) && (40)\\
[\![(*\mathrm{exp}, \mathrm{offs})]\!] &\rightarrow \mathrm{Fat\ temp} = [\![\mathrm{exp}]\!]; \\
&\quad \mathrm{temp.base}.\mathit{read}(\mathrm{temp.offset} + [\![\mathrm{offs}]\!]_t) && (41)
\end{aligned}$$

$$\begin{aligned}
[\![\mathrm{nooffset}]\!]_t &\rightarrow 0 && (42)\\
[\![.\mathrm{id\ offs}]\!]_t &\rightarrow \mathit{offset} + [\![\mathrm{offs}]\!]_t && (43)\\
[\![[\mathrm{exp}]\ \mathrm{offs}]\!]_t &\rightarrow [\![\mathrm{exp}]\!].\mathrm{asInt}() * \mathit{size} + [\![\mathrm{offs}]\!]_t && (44)
\end{aligned}$$

ただし，t は (40)，(41) の右辺のコードで読み出されるオブジェクトの型とする．(40)，(41) の *read* は，t に応じて readFat，readByte，readShort，readLong，readDouble のいずれかとなる．現在の実装では，(40) において id が構造体を参照する場合，または (41) において temp.base が構造体を参照する場合，アクセスメソッドで読み出すことができるデータは構造体を模倣するオブジェクトのみに限定される．したがってこの場合，*read* は read*id* となる（*id* は構造体を模倣するオブジェクトの型である）．(43) は，(40) において id が構造体を参照する場合，または (41) において temp.base が構造体を参照する場合のみ適用される．*offset* は，(43) の左辺の id が示す（構造体の）メンバが格納されているメモリの，メモリブロックの先頭からのオフセットを示す値である．(44) の *size* は，t のサイズを表す整数である．

(40) の右辺は，id が参照する Block オブジェクトの，offs で指定される位置からデータを読み出すことを意味している．(41) の exp はポインタ型である．したがって，(41) の右辺の Java プログラムは，まず exp の変換結果である FatPtr オブジェクトを Fat 型の一時変数に格納する．そして，FatPtr オブジェクトの base が参照する Block オブジェクトの，FatPtr オブジェクトの offset と offs の評価結果を加算した位置から，データを読み出す．

(42)，(43)，(44) は offs の変換規則である．

&lval　変換規則は以下の通りである．

$$\begin{aligned}
[\![\&(\mathrm{id}, \mathrm{offs})]\!] &\rightarrow \textbf{new}\ \mathrm{FatPtr}(\mathrm{id},\ [\![\mathrm{offs}]\!]_t) && (45)\\
[\![\&(*\mathrm{exp}, \mathrm{offs})]\!] &\rightarrow \mathrm{Fat\ temp} = [\![\mathrm{exp}]\!]; \\
&\quad \textbf{new}\ \mathrm{FatPtr}(\mathrm{temp.base},\ \mathrm{temp.offset} + [\![\mathrm{offs}]\!]_t) && (46)
\end{aligned}$$

FatPtr の第一引数は base，第二引数は offset である．

lval = exp　lhost が構造体型でない場合，変換規則は以下の通りである．

$$\begin{aligned}
[\![(\mathrm{id}, \mathrm{offs}) = \mathrm{exp}]\!] &\rightarrow \mathrm{id}.\mathit{write}([\![\mathrm{offs}]\!]_t,\ [\![\mathrm{exp}]\!]) && (47)\\
[\![(*\mathrm{exp1}, \mathrm{offs}) = \mathrm{exp2}]\!] &\rightarrow \mathrm{Fat\ temp} = [\![\mathrm{exp1}]\!]; \\
&\quad \mathrm{temp.base}.\mathit{write}(\mathrm{temp.offset} + [\![\mathrm{offs}]\!]_t,\ [\![\mathrm{exp2}]\!]) && (48)
\end{aligned}$$

ただし (47)，(48) の *write* は，t に応じて writeFat，writeByte，writeShort，writeLong，writeDouble のいずれかとなる．*write* の第一引数は書き込むべき位置，第二引数は書き込むオブジェクトである．

lhost が構造体型の場合，変換規則は以下の通りである．

$$\begin{aligned}
[\![(\mathrm{id}, \mathrm{offs}) = \mathrm{exp}]\!] \quad &\rightarrow \quad \mathrm{id.write}id([\![\mathrm{offs}]\!]_t,\ [\![\mathrm{exp}]\!].\mathrm{duplicate}()) && (49)\\
[\![(*\mathrm{exp1}, \mathrm{offs}) = \mathrm{exp2}]\!] \quad &\rightarrow \quad \mathrm{Fat\ temp} = [\![\mathrm{exp1}]\!];\\
&\qquad \mathrm{temp.base.write}id(\mathrm{temp.offset} + [\![\mathrm{offs}]\!]_t,\\
&\qquad\quad [\![\mathrm{exp2}]\!].\mathrm{duplicate}()) && (50)
\end{aligned}$$

ただし現在の実装では，アクセスメソッドで書き込むことができるデータは構造体を模倣するオブジェクトのみに限定される．したがって，(49)，(50) は write*id* を用いている（*id* は構造体を模倣するオブジェクトの型である）．また，duplicate は構造体のディープコピーを Java で模倣するためのメソッドである．

lval$^?$ = id(exp*)　lval がない場合，変換規則は以下の通りである．

$$\begin{aligned}
[\![\mathrm{id}(\mathrm{exp}^*)]\!] \quad &\rightarrow \quad \textit{blocktype}\ \mathrm{temp1} = \textbf{new}\ \textit{blocktype}(([\![\mathrm{exp1}]\!]);\\
&\qquad \ldots\\
&\qquad \textit{blocktype}\ \mathrm{tempN} = \textbf{new}\ \textit{blocktype}(([\![\mathrm{expN}]\!]);\\
&\qquad \mathrm{id}(\mathrm{temp1},\ \mathrm{temp2},\ \ldots\ \mathrm{tempN}) && (51)
\end{aligned}$$

ただし *blocktype* は，exp の変換結果のオブジェクトが FatInt，Byte，Short，FatLong，Double，FatPtr，*id* であれば，それぞれ FatBlock，ByteBlock，ShortBlock，LongBlock，DoubleBlock，FatBlock，*id*Block となる．

(51) の右辺の Java プログラムは，まずそれぞれの [\![exp]\!] が表すオブジェクトを，新しく作成した要素数が 1 のメモリブロックオブジェクトに格納する．次に，それらに id が参照するメソッドを適用する．

lval がある場合，変換規則は以下の通りである．

$$\begin{aligned}
[\![(\mathrm{id1}, \mathrm{offs}) = \mathrm{id2}(\mathrm{exp}^*)]\!] \quad &\rightarrow \quad \textit{blocktype}\ \mathrm{temp1} = \textbf{new}\ \textit{blocktype}(([\![\mathrm{exp1}]\!]);\\
&\qquad \ldots\\
&\qquad \textit{blocktype}\ \mathrm{tempN} = \textbf{new}\ \textit{blocktype}(([\![\mathrm{expN}]\!]);\\
&\qquad \mathrm{id1}.\textit{write}([\![\mathrm{offs}]\!]_t,\ \mathrm{id2}(\mathrm{temp1},\ \mathrm{temp2},\ \ldots\ \mathrm{tempN})) && (52)\\
[\![(*\mathrm{exp}', \mathrm{offs}) = \mathrm{id}(\mathrm{exp}^*)]\!] \quad &\rightarrow \quad \mathrm{Fat\ temp} = [\![\mathrm{exp}']\!];\\
&\qquad \textit{blocktype}\ \mathrm{temp1} = \textbf{new}\ \textit{blocktype}(([\![\mathrm{exp1}]\!]);\\
&\qquad \ldots\\
&\qquad \textit{blocktype}\ \mathrm{tempN} = \textbf{new}\ \textit{blocktype}(([\![\mathrm{expN}]\!]);\\
&\qquad \mathrm{temp.base.}\textit{write}(\mathrm{temp.offset} + [\![\mathrm{offs}]\!]_t,\\
&\qquad\quad \mathrm{id2}(\mathrm{temp1},\ \mathrm{temp2},\ \ldots\ \mathrm{tempN})) && (53)
\end{aligned}$$

ただし (52)，(53) の *write* は，t に応じて writeFat，writeByte，writeShort，writeLong，writeDouble，write*id* のいずれかとなる（*id* は構造体を模倣するオブジェクトの型である）．

次に，我々は変換規則の適用例を簡単な C プログラム（図 3）で示す．なお，1: といった行番号は説明のために付加されている．

```
 1 :  void main(void){
 2 :    int x;
 3 :    int *p;
 4 :    x = 2;
 5 :    p = &x;
 6 :    while(1){
 7 :      if(*p != 1){}
 8 :      else{break;}
 9 :      x = x - 1;
10 :    }
11 :  }
```

図 3: C プログラム例

まず，変換規則 (3) を適用することにより，プログラムは以下のように変換される．

```
 1 :  void main(string[] args){
 2 :    ⟦int x⟧;
 3 :    ⟦int *p⟧;
 4 :    ⟦x = 2⟧;
 5 :    ⟦p = &x⟧;
6〜10 : ⟦⟦ while(1){ ⟧⟧
          ⟦⟦  ...   ⟧⟧
          ⟦⟦ }      ⟧⟧
11 :  }
```

2 行目は (10)，(13) を適用することにより以下のように変換される．

```
2 :  FatBlock x = new FatBlock(1);
```

3 行目は (10)，(18) を適用することにより以下のように変換される．

```
3 :  FatBlock p = new FatBlock(1);
```

4 行目は (47)，(42)，(20) を適用することにより以下のように変換される．

```
4 :  x.writeFat(0, new FatInt(2));
```

5 行目は (47)，(42)，(45) を適用することにより以下のように変換される．

```
5 :  p.writeFat(0, new FatPtr(x, 0));
```

6〜10 行目は，(31) を適用することにより以下のように変換される．

```
 6 :  while(true){
7〜8 :   ⟦if ...⟧
 9 :    ⟦x = x - 1⟧;
10 :  }
```

```
 1 :  void main(string[] args){
 2 :    FatBlock x = new FatBlock(1);
 3 :    FatBlock p = new FatBlock(1);
 4 :    x.writeFat(0, new FatInt(2));
 5 :    p.writeFat(0, new FatPtr(x, 0));
 6 :    while(true){
          Fat temp = p.readFat(0);
 7 :      if((new FatInt(temp.base.readFat(temp.offset + 0).asInt()
             != new FatInt(1).asInt() ? 1 : 0).asInt() != 0){}
 8 :      else{break;}
 9 :      x.writeFat(0, new FatInt(x.readFat(0).asInt() − new FatInt(1).asInt()))
10 :    }
11 :  }
```

図 4: 変換後の Java プログラム

7，8 行目は，(30)，(28) を適用することにより以下のように変換される．

```
7 :  if([[*p != 1]].toobj() != 0){}
8 :  else{break;}
```

[[*p != 1]] は，(26)，(20) を適用することにより以下のように変換される．

```
new FatInt([[*p]].toobj() != new FatInt(1).asInt() ? 1 : 0)
```

[[*p]] は，(41)，(42)，(40) を適用することにより以下のように変換される．

```
Fat temp = p.readFat(0);
temp.base.readFat(temp.offset + 0)
```

したがって，7，8 行目は最終的に以下のようなプログラムに変換される．

```
    Fat temp = p.readFat(0);
7 : if((new FatInt(temp.base.readFat(temp.offset + 0).asInt()
        != new FatInt(1).asInt() ? 1 : 0).asInt() != 0){}
8 : else{break;}
```

9 行目は，(47)，(42)，(24)，(40)，(20) を適用することにより以下のように変換される．

```
9 :  x.writeFat(0, new FatInt(x.readFat(0).asInt() − new FatInt(1).asInt()));
```

結果出力される Java プログラムは図 4 の通りである．