推薦論文● PPL 2007

Java 言語への変換によるポインタ演算の安全な実装方式

上嶋 祐紀　住井 英二郎

C 言語サブセットのプログラムを安全な Java 言語のプログラムに変換する方式を実装した．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，まず 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．次に，これらのクラスを利用するような Java への変換規則を定め，規則に従ってトランスレータを実装した．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，などの最適化を実装した．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 3.3 倍〜585 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）1.3 倍〜5.9 倍程度に改善した．

We implemented a translator from a subset of C to Java that guarantees safety. For such translation, we need to emulate C's pointer arithmetics in Java. We thus defined Java classes to represent C's pointers and memory blocks. Next, we defined translation rules which use these classes, and implemented a translator following the rules. We also need to translate integers as pointers, because they can be cast to each other in C. However, representing all integers as pointers incurs too much overhead. We therefore implemented optimizations, including a data flow analysis for translating C's integer variables (to which no pointers are assigned) to Java's integer variables. We conducted experiments with 9 benchmark programs. Without optimizations, the translated programs were 3.3–585 times slower than the original C programs. After the optimizations, those numbers improved to 1.3–5.9.

1　序論

一般にプログラミング言語の安全性 (safety) 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．本論文におけるプログラミング言語の安全性とは以下の意味である [11, 13]．まず，プログラミング言語の意味論は，**一ステップごとの簡約関係**(small-step reduction relation) $\rightharpoonup$ により表現できるとする．$\rightharpoonup$ はプログラムの実行状態の遷移系とみなすことができる．P が終了状態ではなく，かつ，$P \rightharpoonup P'$ なる P' が存在しないとき，P は stuck であるという．終了状態は言語ごとに定義されるものとする．（静的型検査を含め構文論的に合法な）任意のプログラムの初期状態 P に対し，$P \rightharpoonup^* P'$ なる任意の P' が stuck でないとき，言語 $\rightharpoonup$ は**安全**(safe) であるという．ただし，$\rightharpoonup^*$ は $\rightharpoonup$ の反射的推移的閉包である．

C 言語は，現在も広く使われているプログラミング言語の一つであるが，安全性が保証されていない．（C 言語には標準では形式的意味論が存在しないが，非形式的な標準規格 [1] では）多数の動作が**未定義**(undefined) とされており，一ステップごとの簡約関係で意味論を定式化したとすれば，stuck 状態に簡約されるプログラムが存在すると想定されるためである．C 言語の未定義動作は，予期せぬ結果やセキュリティホールの原因となっている．（なお，C 言語には関

Safe implementation of C pointer arithmetics by translation to Java.

Yuhki Kamijima, Eijiro Sumii, 東北大学大学院情報科学研究科, Graduate School of Information Sciences, Tohoku University.

コンピュータソフトウェア, Vol.xx, No.y (zzzz), pp.vv.
[推薦論文] xxxx 年 y 月 z 日受付.

数の実引数の評価順序など，**未規定**(unspecified) とされる動作もあるが，それら自体は上述の意味で安全である．未規定の動作は，$P' \rightharpoonup P''$ なる複数の P'' が存在するケースに当たり，P' が stuck であることとは異なるからである．同じ理由で，例えば Scheme 言語は安全と考えられる.）

そこで，我々は Java 言語の（ネイティブメソッドなどを除いた）主な部分が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 言語のプログラムに変換することにより，C 言語の安全な処理系を実現する手法を検討している．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（B 節参照）

というものである [14]．C のプログラムを Java へ変換することにより，Java 処理系の安全性を仮定すれば，C プログラムの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．本論文では，C 言語のメモリモデルとポインタ演算を Java 言語に変換する手法について議論する．

我々はまずポインタ演算を 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 [2, 14] 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 ワード目はポインタの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．これを Java の FatPtr クラスとして実装した．

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 [9] 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．

以上のような変換はプログラマの手で行うこともできるが，Java にポインタ演算がない以上，いずれにせよ一般には同様の概念が必要となる．

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 [2] および Fail-Safe C [9, 14] に由来するものである．本研究では，コンパイラの生成する出力を低水準なコードではなく,（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単純かつ安全に実現する．

本論文の以降の構成は次の通りである．2 節では本研究と既存研究の関連を議論する．3 節では，変換に利用するクラスの実装について述べる．4 節では，基本的な変換の方式について直観的に述べる．5 節では，変換元言語・変換先言語・変換規則の核となる部分の形式的定義を与え，その部分について変換の正しさを示す．6 節では，変換結果のプログラムを高速化するための最適化について述べる．7 節では比較的単純なベンチマークプログラムによる実験結果を示し，8 節では今後の研究課題について述べる．

2 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 [4, 6]．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，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 [8] 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一部対応していない．

```
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

Fail-Safe C [9, 14] 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．3 節で述べる 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 の概念をより単純かつ安全に実現している．

3 変換に利用する Java のクラス

1 節で述べたように，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，ポインタを整数とみな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．

```
abstract class Block {
    int objsize;
    int size;
    int addr;
    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 {
    private 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

NULL は base が null で offset が 0 の FatPtr とする．

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 [14]．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．（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考えられるが，HotSpot VM の内部動作が観察困難なため，具体的な検証には至っていない．）

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，int 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．具体的には，byteBlock の objsize（char のサイズ）は 1，shortBlock は 2，FatBlock は 4，doubleBlock は 8 としている．

なお，現在の実装では，AddrCounter の整数オーバーフローについては考慮していない．したがって，AddrCounter がオーバーフローすると，ポインタを整数にキャストして比較した結果が矛盾することがありうる．たとえば，確率は低いが，相異なるポインタを整数にキャストした値が等しくなることもありうる．ただし，仮に AddrCounter がオーバーフローにより重複しても，生成される Block オブジェクトそのものは異なるので，問題はポインタを整数にキャストして比較する場合のみであり，異なるメモリブロックへのアクセスが混同されることはない．

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したり，境界調整されていない（x のサイズの倍数になっていない）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．

4 基本的な変換法

3 節で述べたクラスを利用して，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．

なお，Block オブジェクトは Java のヒープに確保されるため，その生存期間は実質無限となる．したがって，自動変数の宣言を単純に Block オブジェクトの生成に変換すると，生存期間が終了した自動変数へのポインタ経由アクセス（仕様では未定義動作）もエラーにならない．それだけでなく，大きな（ないし多くの）記憶領域へのポインタだけが残るようなプログラムでは，メモリリークのおそれもある．そこで我々の実装では，生存期間の終了と同時に Block オブジェクトの contents フィールドに null を代入することにより，C 言語の自動変数と同様の生存期間を実現する．

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

$$
\begin{array}{rcl}
v & ::= & \mathrm{num}(n) \mid \mathrm{ptr}(b, \mathit{off}) \\
t & ::= & \mathrm{int} \mid t\ \mathrm{pointer} \\
e & ::= & v \mid x \mid *e \mid *e = e; e \\
 & \mid & (t)e \mid e + e \mid e \oplus e \\
 & \mid & \mathrm{malloc}(e) \mid \mathrm{let}\ x = e\ \mathrm{in}\ e
\end{array}
$$

図 3 C サブセットの抽象構文

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．(加算 + や減算 − など，型ごとにオーバーロードされている演算は，静的型検査時に区別できるので，混同のおそれはない．)

構造体（たとえば struct xyz）は，各々のフィールドをメンバとする Java のクラス Xyz と，構造体の配列を格納するための XyzBlock クラスに変換する．構造体メンバへのポインタアクセスは，現在は実装されていないが，Fail-Safe C [9] と同様にアクセスメソッドにより実現できる．

5 抽象構文と変換規則

4 節で述べた基本的な変換法を定式化するために，我々はまず C サブセットの抽象構文（図 3）と Java サブセットの抽象構文（図 4）を定義した．変換においてもっとも本質的である Fat ポインタとメモリブロックをできるだけ単純に定式化するため，値としては int 型とポインタ型のみを考慮した．また，境界調整の問題は割愛し，いずれの値もサイズは 1 とした．

C サブセットの抽象構文は，値 v として整数 $\mathrm{num}(n)$ とポインタ $\mathrm{ptr}(b, \mathit{off})$（$b$ はメモリブロックの先頭アドレスを表す整数，off はそのメモリブロック内でのオフセットを意味する整数）を持つ [10]．また，式 e として値 v，変数 x，ポインタを介した値の読み出し $*e$，ポインタを介した値の書き込み $*e = e; e$，int 型あるいはポインタ型へのキャスト $(t)e$，整数加算 $e + e$，ポインタ加算 $e \oplus e$，メモリブロックの確保 $\mathrm{malloc}(e)$，局所定義 $\mathrm{let}\ x = e\ \mathrm{in}\ e$ を持つ．

$$
\begin{array}{rcl}
w & ::= & n \mid l \mid \mathrm{null} \\
 & \mid & \mathrm{new\ FatPtr}(\mathrm{null}, n) \\
 & \mid & \mathrm{new\ FatPtr}(l, n) \\
d & ::= & w \mid y \mid \mathrm{new\ FatPtr}(d, d) \\
 & \mid & \mathrm{new\ FatBlock}(d) \\
 & \mid & d.\mathrm{read}(d) \mid d.\mathrm{write}(d, d); d \\
 & \mid & d.\mathrm{base} \mid d.\mathrm{offset} \mid d.\mathrm{asInt}() \\
 & \mid & d + d \mid \mathrm{let}\ y = d\ \mathrm{in}\ d
\end{array}
$$

図 4 Java サブセットの抽象構文

Java サブセットの抽象構文は，値 w として整数 n，参照 l，null，FatInt オブジェクト new FatPtr(null, n)，FatPtr オブジェクト new FatPtr(l,n) を持つ．また，式 d として値 w，変数 y，FatPtr オブジェクトの生成 new FatPtr(d,d)，FatBlock オブジェクトの生成 new FatBlock(d)，FatBlock からの値の読み出し d.read(d)，FatBlock への値の書き込み d.write(d,d);d，FatPtr の base フィールドの参照 d.base，offset フィールドの参照 d.offset，asInt メソッドの呼び出し d.asInt()，整数加算 $d + d$，局所定義 $\mathrm{let}\ y = d\ \mathrm{in}\ d$ を持つ．

これらの定義を基に，C から Java への変換規則（図 5）を定義した．ただし，$[\![]\!]_\epsilon$ は変換を表す記号である．また，ϵ は整数の集合から参照の集合への部分単射とする．これは，C でのアドレスに対応する Java の参照を定めるために用いられる．$\mathrm{num}(n)$ は FatInt オブジェクトに，$\mathrm{ptr}(b, \mathit{off})$ は FatPtr オブジェクトに変換される (trans-num, trans-ptr)．$*e$ は，FatPtr の baseが参照する FatBlock の，(FatPtr の）offset で示される位置から値を読み出す式に変換される (trans-deref)．$*e = e; e$ も同様に，値を書き込む式に変換される (trans-update)．$(t)e$ は，キャ

$[\![\mathrm{num}(n)]\!]_\epsilon$	=	new FatPtr(null, n)	(trans-num)
$[\![\mathrm{ptr}(b, \mathit{off})]\!]_\epsilon$	=	new FatPtr($\epsilon(b)$, off)	(trans-ptr)
$[\![x]\!]_\epsilon$	=	x	(trans-var)
$[\![*e]\!]_\epsilon$	=	let $y = [\![e]\!]_\epsilon$ in y.base.read(y.offset)	(trans-deref)
$[\![*e_1 = e_2; e_3]\!]_\epsilon$	=	let $y = [\![e_1]\!]_\epsilon$ in y.base.write(y.offset, $[\![e_2]\!]_\epsilon$); $[\![e_3]\!]_\epsilon$	(trans-update)
$[\![(t)e]\!]_\epsilon$	=	$[\![e]\!]_\epsilon$	(trans-cast)
$[\![e_1 + e_2]\!]_\epsilon$	=	new FatPtr(null, $[\![e_1]\!]_\epsilon$.asInt() + $[\![e_2]\!]_\epsilon$.asInt())	(trans-int-add)
$[\![e_1 \oplus e_2]\!]_\epsilon$	=	let $y = [\![e_1]\!]_\epsilon$ in	
		new FatPtr(y.base, y.offset + $[\![e_2]\!]_\epsilon$.asInt())	(trans-ptr-add)
$[\![\mathrm{malloc}(e)]\!]_\epsilon$	=	new FatPtr(new FatBlock($[\![e]\!]_\epsilon$.asInt()), 0)	(trans-alloc)
$[\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e_2]\!]_\epsilon$	=	let $x = [\![e_1]\!]_\epsilon$ in $[\![e_2]\!]_\epsilon$	(trans-let)

図 5　C から Java への変換規則

スト (t) を無視し単に式 e として変換される (trans-cast)．$e + e$ は，asInt メソッドを用いた式に変換される (trans-int-add)．$e \oplus e$ は，FatPtr の *off* に整数を加算する式に変換される (trans-ptr-add)．malloc(e) は，新たに生成した FatBlock への参照を base に持つ FatPtr を生成する式に変換される (trans-alloc)．

以上の定義を用いて，我々は変換の正しさを証明した（A 節）．正しさを表す定理の概要は以下の通りである．

- 式 e が e' に 1 ステップで簡約されるならば，式 $[\![e]\!]_\epsilon$ は $[\![e']\!]_\epsilon$ に 0 ステップ以上で簡約される．
- 式 e が stuck 状態ならば，式 $[\![e]\!]_\epsilon$ も 0 ステップ以上の簡約により stuck 状態となる．（実際の Java 言語では stuck 状態ではなく，例外が発生するので安全である．）

6　最適化

4 節で述べた基本的な変換法や，5 節で述べた変換規則では，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，あるいは配列型の変数は，すべて 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を生じる．

これらのオーバーヘッドを削減するために，アドレス演算子 (&) によりアドレスを取られることがない C の基本型変数（および要素のアドレスを取られることがない配列型変数）は Block オブジェクトとせず，以下のように変換する．int 型あるいはポインタ型の変数は，Fat 型の変数に変換する（int 配列型あるいはポインタ配列型の変数は，Fat 配列型の変数に変換する）．long int 型も int 型と同様に扱う．long long int 型は，offset フィールドを Java の long 型とした FatLong クラスとする．それ以外の基本型変数は，Java の通常の基本型変数に変換する（同様に，それ以外の基本型を要素型とする配列型変数は，Java の通常の基本型を要素型とする配列型変数に変換する）．

加えて，アドレスを取られない int 型変数および int 配列型変数については，以下の最適化を行う．標準的な関数内データフロー解析 [5] により，ポインタを代入されることがない，またはポインタとして使用されることがない int 型変数を求め，それらを Java の通常の int 型変数に変換する（配列についても，要素にポインタを代入されることがない，または要素がポインタとして使用されることがない int 配列型変数は，Java の通常の int 配列型変数に変換する）．

さらに，C の int 型の値は Java の int 型の値に変換し，必要になるまで FatInt 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．

- 構造体のメンバ，FatBlock オブジェクトに変換

された変数，ないし FatBlock オブジェクトに変換された配列の要素に代入される．

- 関数の実引数または返り値になる．

また，4 節ではすべての関数の引数を Block クラスのオブジェクトに変換していたので（関数の中で仮引数のアドレスを取られる可能性があるため），それらの引数がループや再帰関数の中でアクセスされるとオーバーヘッドを生じていた．このオーバーヘッドを削減するために，我々は一般に関数定義 f(t x) {...} を f(t x') {t x = x'; ...} のように書き換えることとした．ただし x' は他に出現しない新しい変数である．これにより，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，先の条件を満たせば，x を（doubleBlock ではなく double などの）基本型の変数に最適化することも可能になる．

これらの最適化により，7 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．

7 実験結果と考察

我々は，4 節で述べた基本的な変換法と 5 節の変換規則，および 6 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 [7] 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 [3] を利用した．データフロー解析には CIL の Dataflow モジュールを用いた．

その上で，4 節で述べた変換法および 6 節で述べた最適化を評価するために，The Computer Language Shootout Benchmarks[†1] より 9 個の C プログラム（表 1）を用いて実験を行った．これらは数十行程度の小規模なプログラムだが，整数演算やポインタ演算を含み，提案手法のオーバーヘッドを測定・分析するという目的において，ある程度は有用であると考えられる．特に 6 節の最適化は関数内解析のみ行っているため，個々の関数のサイズやプログラミングスタイルが変わらない限り，仮に関数の個数だけ増えても解析精度に直接の影響はない．

†1 http://shootout.alioth.debian.org

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 1.6.0 を用い，Java 仮想マシンには-server オプションをつけて実行した．また，比較のための C コンパイラとして GCC 4.2.2 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，（3 節の Java クラスを使わずに）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である．

なお，recursive の unopt は，デフォルトのスタックサイズ（320 KB）と最大ヒープサイズ（64 MB）では実行できなかったため，スタックサイズを 256 MB，最大ヒープサイズを 1024 MB に変更して実行した．同様に recursive の opt と hand-written は，スタックサイズを 512 KB にして実行した．binary-trees は，unopt, opt, hand-written のすべてについて，最大ヒープサイズを 128 MB として実行した．

表 2 において，nsieve, mandelbrot, partial-sums と nsieve-bits では，optimized と hand-written の値にほとんど差がない．これは，6 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[†2]．

1. recursive, binary-trees では，関数の引数が Fat オブジェクトとなり，ループや再帰関数の中で

†2 fannkuch の optimized と hand-written の差の原因は不明である．このプログラムのみ，ほとんど同一のバイトコードであっても実行時間の誤差が大きい．

表 1　ベンチマークプログラムの概要

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．ポインタ演算は含まれない．
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．ポインタを用いて配列の要素の読み書きを行う．
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．ポインタ演算は含まれない．
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．ポインタ演算は含まれない．
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．ポインタ演算は含まれない．
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．ポインタ演算は含まれない．
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．ポインタ演算は含まれない．
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．構造体へのポインタを関数間で受け渡しする．
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．ポインタを用いて配列の要素の読み出しを行う．

表 2　性能評価実験の結果（GCC による実行時間を 1 とする）

	nsieve	spectral-norm	mandelbrot	recursive
unoptimized	8.55	15.90	6.55	584.87
optimized	1.33	3.31	3.30	4.37
hand-written	1.31	0.91	3.28	1.08

	partial-sums	nsieve-bits	fannkuch	binary-trees	n-body
unoptimized	3.25	17.16	23.01	21.02	6.28
optimized	2.45	3.05	2.06	5.86	2.30
hand-written	2.45	3.04	1.94	0.91	1.20

アクセスされることによるオーバーヘッドが大きい．

2. spectral-norm, binary-trees, 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．

1 番目の問題については，手続き間解析を用いるか，関数を（int 版と Fat 版に）複製するなどして，int 型の引数をできるだけ Fat ではなく通常の Java の int に変換する必要がある．一方，2 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 の参照に変換するなど，（典型的ではあるが）より複雑な解析・最適化が必要となる．

8 結論と今後の課題

本論文では，ポインタ演算を安全に模倣することにより，C 言語のサブセットを Java へ変換する手法を提案した．

本研究の今後の方向の一つに，C 言語のより大きなサブ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 C 言語標準仕様のうち，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考えられる．

多次元配列 Java にも多次元配列はあるが，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 ループからの脱出や繰り返しを飛ばすために用いられている C の goto 文は，Java のラベル付き break 文およびラベル付き continue 文に変換できる．しかし，それ以外の一般の goto 文については，より大掛かりなプログラム変換（一種の CPS 変換 [12] など）が必要になると思われる．

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，検証が必要である．また，7 節のプログラムはいずれも数十行程度と小規模であるため，より大規模なプログラムにおいて，プログラミングスタイルなどの定性的変化による性能への影響がないかどうかも検証が必要である．一つの目標として，SPEC CPU ベンチマーク（の大半）を動作させ，性能評価実験を行うことを目指したいと考えている．

7 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），JIT コンパイラの内部動作が観察しづらいという困難はあるが，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，安全な言語として広く用いられている 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，PPL 2007 の査読者の方々，コンピュータソフトウェア誌の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] ：プログラム言語 C, JIS X 3010:2003 (ISO/IEC 9899:1999).

[2] Austin, T. M., Breach, S. E., and Sohi, G. S.: Efficient Detection of All Pointer and Array Access Errors, *Programming Language Design and Implementation*, 1994, pp. 290–301.

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

[4] ：Jazillian: The Legacy to “Natural” Java Translator. `http://www.jazillian.com`.

[5] Kildall, G. A.: A unified approach to global pro-

gram optimization, *Proceedings of the 1st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1973, pp. 194–206.
[6] Martin, J. and Muller, H.: Strategies for Migration from C to Java, *Software Maintenance and Reengineering*, 2001, pp. 200–209.
[7] Necula, G., McPeak, S., Rahul, S. P., and Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, *Compiler Construction*, 2002, pp. 213–228.
[8] Necula, G., McPeak, S., and Weimer, W.: CCured: Type-Safe Retrofitting of Legacy Code, *Principles of Programming Languages*, 2002, pp. 128–139.
[9] Oiwa, Y.: *Implementation of a Fail-Safe ANSI C Compiler*, PhD Thesis, University of Tokyo, 2004.
[10] Oiwa, Y., Sekiguchi, T., Sumii, E., and Yonezawa, A.: Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure (Progress Report), *Software Security – Theories and Systems*, 2002, pp. 133–153.
[11] Pierce, B. C.: *Types and Programming Languages*, MIT Press, 2002.
[12] Strachey, C. and Wadsworth, C. P.: Continuations: A mathematical semantics for Handling Full Jumps, Technical Report PRG-11, Programming Research Group Technical Monograph, Oxford University Computing Laboratory, 1974. Reprinted in *Higher-Order and Symbolic Computation*, vol. 13, no. 1–2, pp. 135–152, 2000.
[13] Wright, A. K. and Felleisen, M.: A syntactic approach to type soundness, *Information and Computation*, Vol. 115,No. 1(1994), pp. 38–94.
[14] 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 変換の正しさの証明

5 節の変換規則の正しさを示す．そのために，まず C 言語におけるヒープ H を，整数 b の集合から配列 $[v_0, \ldots, v_m]$ の集合への部分写像と定義する [10]．また，Java 言語におけるヒープ I を，参照 l の集合から

$$ob \quad ::= \quad \mathrm{FatBlock}(\mathrm{a}, [w_0, \ldots, w_m])$$

の集合への部分写像と定義する．ただし a は（AddrCounter から計算される）整数である．

さらに，ヒープの変換規則を図 6 のように定義する．

次に，C サブセットの簡約規則（図 7）と Java サブセットの簡約規則（図 8）を定義する．記号の定義は以下の通りである [10].

- $\rightarrow^*$: $\rightarrow$ の反射的推移的閉包
- $H; e \nrightarrow$: $H; e \rightarrow H'; e'$ となるような H' と e' が存在しない
- $H\{b := \ldots\}$: 部分写像 H に対し，b の値を $\ldots$ に変更した部分写像
- $e[v/x]$: e 中の全ての自由変数 x に値 v を代入した式
- $v \times n$: 配列 $[v_1, \ldots, v_n]$ ただし $v = v_1 = \ldots = v_n$
- off inside $H(b)$: $H(b) = [v_0, \ldots, v_m]$ として $0 \leq off \leq m$
- $\min(b)\{\ldots\}$: $\{\ldots\}$ 内の条件を満たす最小の b
- $\mathrm{addr}(H)$: $\min(b)\{\forall b' \in \mathrm{dom}(H).$
  $\forall off' \text{ inside } H(b').\ (b' + off') < b\}$

また，C サブセットと Java サブセットそれぞれの簡約関係の直観的意味は以下の通りである．

- $H; e \rightarrow H'; e'$: ヒープ H のもとで，式 e は e' に 1 ステップで簡約され，H は H' に更新される
- $I, \mathrm{A}; d \rightarrow I', \mathrm{A}'; d'$: ヒープ I と AddrCounter A のもとで，式 d は d' に 1 ステップで簡約され，I は I' に，A は A′ に更新される

これらの簡約規則と変換規則，さらに以下の補題 1，2，3 を用いて，変換の正しさを証明する．

補題 1

$[\![H]\!]_\epsilon, \mathrm{addr}(H); [\![e_0]\!]_\epsilon \rightarrow [\![H']\!]_{\epsilon'}, \mathrm{addr}(H'); [\![e_0']\!]_{\epsilon'}$ ならば

$$\forall C.\ [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![C[e_0]]\!]_\epsilon \rightarrow^* [\![H']\!]_{\epsilon'}, \mathrm{addr}(H'); [\![C[e_0']]\!]_{\epsilon'}$$

証明． C の構文に関する帰納法による． □

補題 2 $e = C[e_0]$ かつ $C \neq []$ かつ e_0 が値でないかつ $H; e_0 \nrightarrow$ ならば

$\forall \epsilon.\ \exists I, \mathrm{A}, d.\ [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![e]\!]_\epsilon \rightarrow^* I, \mathrm{A}; d \nrightarrow$ かつ d は値でない．

証明． C の構文に関する帰納法による． □

補題 3

$$[\![e[v/x]]\!]_\epsilon = [\![e]\!]_\epsilon[[\![v]\!]_\epsilon/x]$$

証明． e の構文に関する帰納法による． □

変換の正しさとして，以下の定理 1，2 が共に成り立つことを示す．

定理 1 $H; e \rightarrow H'; e'$ ならば

$\forall \epsilon.\ \exists \epsilon'.$

$[\![H]\!]_\epsilon, \mathrm{addr}(H); [\![e]\!]_\epsilon \rightarrow^* [\![H']\!]_{\epsilon'}, \mathrm{addr}(H'); [\![e']\!]_{\epsilon'}$

$$[\![H]\!]_\epsilon \quad = \quad I \text{ where } \mathrm{dom}(I) = \{\epsilon(b) \mid b \in \mathrm{dom}(H)\} \text{ and}$$
$$\forall b.\ H(b) = [v_0, \ldots, v_m] \Rightarrow I(\epsilon(b)) = \mathrm{FatBlock}(b, [[\![v_0]\!]_\epsilon, \ldots, [\![v_m]\!]_\epsilon]) \quad \text{(trans-heap)}$$

図 6 ヒープの変換規則

定理 2 $H; e \nrightarrow$ かつ e が値でなければ，

$\forall \epsilon.\ \exists I, \mathrm{A}, d.\ [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![e]\!]_\epsilon \rightarrow^* I, \mathrm{A}; d \nrightarrow$ かつ d は値でない． □

まず定理 1 を，$H; e \rightarrow H'; e'$ の導出に関する帰納法で証明する．導出の最後に使われた簡約規則によって場合分けをする．

最後に使われた規則が (C-deref) の場合，$H(b) = [v_0, \ldots, v_{off}, \ldots, v_m]$，$e = *(\mathrm{ptr}(b, off))$，$H' = H$，$e' = v_{off}$ とする．(trans-heap) より，

$[\![H]\!]_\epsilon(\epsilon(b)) =$
$\quad \mathrm{FatBlock}(b, [[\![v_0]\!]_\epsilon, \ldots, [\![v_{off}]\!]_\epsilon, \ldots, [\![v_m]\!]_\epsilon])$

また，(trans-deref)，(trans-ptr) および (Java-let)，(Java-base)，(Java-offset)，(Java-read) より，

$$\begin{aligned} & [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![*(\mathrm{ptr}(b, off))]\!]_\epsilon \\ = \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\ & \text{let } y = \text{new FatPtr}(\epsilon(b), off) \text{ in} \\ & y.\mathrm{base.read}(y.\mathrm{offset}) \\ \rightarrow^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![v_{off}]\!]_\epsilon \end{aligned}$$

したがって，$\epsilon' = \epsilon$ と取れば良い．

最後に使われた規則が (C-update) の場合，$H(b) = [v_0, \ldots, v_{off}, \ldots, v_m]$，$e$ は $*(\mathrm{ptr}(b, off)) = v; e_0$，$H' = H\{b := [v_0, \ldots, v, \ldots, v_m]\}$，$e' = e_0$ とする．(trans-heap) より，

$[\![H]\!]_\epsilon(\epsilon(b)) =$
$\quad \mathrm{FatBlock}(b, [[\![v_0]\!]_\epsilon, \ldots, [\![v_{off}]\!]_\epsilon, \ldots, [\![v_m]\!]_\epsilon])$

$[\![H']\!]_\epsilon(\epsilon(b)) =$
$\quad \mathrm{FatBlock}(b, [[\![v_0]\!]_\epsilon, \ldots, [\![v]\!]_\epsilon, \ldots, [\![v_m]\!]_\epsilon])$

また，(trans-update)，(trans-ptr) および (Java-let)，(Java-base)，(Java-offset)，(Java-write) より，

$$\begin{aligned} & [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![*(\mathrm{ptr}(b, off)) = v; e_0]\!]_\epsilon \\ = \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\ & \text{let } y = \text{new FatPtr}(\epsilon(b), off) \text{ in} \\ & y.\mathrm{base.write}(y.\mathrm{offset}, [\![v]\!]_\epsilon); [\![e_0]\!]_\epsilon \\ \rightarrow^* \quad & [\![H']\!]_\epsilon, \mathrm{addr}(H); [\![e_0]\!]_\epsilon \end{aligned}$$

したがって，$\epsilon' = \epsilon$ と取れば良い．

最後に使われた規則が (C-cast-num) または (C-cast-ptr) の場合は，(trans-cast)，(trans-num)，および (trans-ptr) により容易に証明できる．

最後に使われた規則が (C-int-add1) の場合，$H = H'$，$e = \mathrm{num}(n_1) + \mathrm{num}(n_2)$，$e' = \mathrm{num}(n_1 + n_2)$ とする．(trans-int-add)，(trans-num) および (Java-asInt-null) より，

$$\begin{aligned} & [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![\mathrm{num}(n_1) + \mathrm{num}(n_2)]\!]_\epsilon \\ = \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\ & \text{new FatPtr(null,} \\ & \quad \text{new FatPtr(null, } n_1\text{).asInt() +} \\ & \quad\quad \text{new FatPtr(null, } n_2\text{).asInt())} \\ \rightarrow^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \text{new FatPtr(null, } n_1 + n_2\text{)} \\ = \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![\mathrm{num}(n_1 + n_2)]\!]_\epsilon \end{aligned}$$

したがって，$\epsilon' = \epsilon$ と取れば良い．

最後に使われた規則が (C-int-add2) の場合，$b_1 \in \mathrm{dom}(H)$，$H = H'$，$e = \mathrm{ptr}(b_1, off_1) + \mathrm{num}(n_2)$，$e' = \mathrm{num}(b_1 + off_1 + n_2)$ とする．(trans-int-add)，(trans-ptr)，(trans-num) および (Java-asInt)，(Java-asInt-null) より，

$$\begin{aligned} & [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![\mathrm{ptr}(b_1, off_1) + \mathrm{num}(n_2)]\!]_\epsilon \\ = \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\ & \text{new FatPtr(null,} \\ & \quad \text{new FatPtr}(\epsilon(b_1), off_1\text{).asInt() +} \\ & \quad\quad \text{new FatPtr(null, } n_2\text{).asInt())} \\ \rightarrow^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\ & \text{new FatPtr(null, } b_1 + off_1 + n_2\text{)} \\ = \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![\mathrm{num}(b_1 + off_1 + n_2)]\!]_\epsilon \end{aligned}$$

したがって，$\epsilon' = \epsilon$ と取れば良い．

(C-int-add3) と (C-int-add4) の場合は，(C-int-add2) の場合と同様である．

最後に使われた規則が (C-ptr-add1) の場合，$H = H'$，$e = \mathrm{ptr}(b_1, off_1) \oplus \mathrm{num}(n_2)$，$e' = \mathrm{ptr}(b_1, off_1 + n_2)$ とする．(trans-ptr-add)，(trans-ptr)，(trans-num) および (Java-let)，(Java-base)，(Java-offset)，

$$
\begin{array}{rcl}
C & ::= & [] \mid *C \mid *C = e; e \mid *v = C; e \mid *v = v; C \mid (t)C \\
& \mid & C + e \mid v + C \mid C \oplus e \mid v \oplus C \mid \mathrm{malloc}(C) \\
& \mid & \mathrm{let}\ x = C\ \mathrm{in}\ e \mid \mathrm{let}\ x = v\ \mathrm{in}\ C
\end{array}
$$

$$\frac{H(b) = [v_0, \ldots, v_{off}, \ldots, v_m]}{H; *(\mathrm{ptr}(b, off)) \to H; v_{off}} \quad \text{(C-deref)}$$

$$\frac{H(b) = [v_0, \ldots, v_{off}, \ldots, v_m]}{H; *(\mathrm{ptr}(b, off)) = v; e \to H\{b := [v_0, \ldots, v, \ldots, v_m]\}; e} \quad \text{(C-update)}$$

$$H; (t)(\mathrm{num}(n)) \to H; \mathrm{num}(n) \quad \text{(C-cast-num)}$$

$$H; (t)(\mathrm{ptr}(b, off)) \to H; \mathrm{ptr}(b, off) \quad \text{(C-cast-ptr)}$$

$$H; \mathrm{num}(n_1) + \mathrm{num}(n_2) \to H; \mathrm{num}(n_1 + n_2) \quad \text{(C-int-add1)}$$

$$\frac{b_1 \in \mathrm{dom}(H)}{H; \mathrm{ptr}(b_1, off_1) + \mathrm{num}(n_2) \to H; \mathrm{num}(b_1 + off_1 + n_2)} \quad \text{(C-int-add2)}$$

$$\frac{b_2 \in \mathrm{dom}(H)}{H; \mathrm{num}(n_1) + \mathrm{ptr}(b_2, off_2) \to H; \mathrm{num}(n_1 + b_2 + off_2)} \quad \text{(C-int-add3)}$$

$$\frac{b_1, b_2 \in \mathrm{dom}(H)}{H; \mathrm{ptr}(b_1, off_1) + \mathrm{ptr}(b_2, off_2) \to H; \mathrm{num}(b_1 + off_1 + b_2 + off_2)} \quad \text{(C-int-add4)}$$

$$H; \mathrm{ptr}(b_1, off_1) \oplus \mathrm{num}(n_2) \to H; \mathrm{ptr}(b_1, off_1 + n_2) \quad \text{(C-ptr-add1)}$$

$$\frac{b_2 \in \mathrm{dom}(H)}{H; \mathrm{ptr}(b_1, off_1) \oplus \mathrm{ptr}(b_2, off_2) \to H; \mathrm{ptr}(b_1, off_1 + b_2 + off_2)} \quad \text{(C-ptr-add2)}$$

$$H; \mathrm{num}(n_1) \oplus \mathrm{num}(n_2) \to H; \mathrm{num}(n_1 + n_2) \quad \text{(C-ptr-add3)}$$

$$\frac{b_2 \in \mathrm{dom}(H)}{H; \mathrm{num}(n_1) \oplus \mathrm{ptr}(b_2, off_2) \to H; \mathrm{num}(n_1 + b_2 + off_2)} \quad \text{(C-ptr-add4)}$$

$$\frac{b = \mathrm{addr}(H) \qquad n > 0}{H; \mathrm{malloc}(\mathrm{num}(n)) \to H\{b := \mathrm{num}(0) \times n\}; \mathrm{ptr}(b, 0)} \quad \text{(C-alloc-num)}$$

$$\frac{b \in \mathrm{dom}(H)}{H; \mathrm{malloc}(\mathrm{ptr}(b, off)) \to H; \mathrm{malloc}(\mathrm{num}(b + off))} \quad \text{(C-alloc-ptr)}$$

$$H; \mathrm{let}\ x = v\ \mathrm{in}\ e \to H; e[v/x] \quad \text{(C-let)}$$

$$\frac{H; e \to H'; e'}{H; C[e] \to H'; C[e']} \quad \text{(C-context)}$$

図 7　C サブセットの簡約規則

$$
\begin{array}{rcl}
D & ::= & [] \mid \text{new FatPtr}(D, d) \mid \text{new FatPtr}(w, D) \mid \text{new FatBlock}(D) \\
& \mid & D.\text{read}(d) \mid w.\text{read}(D) \mid D.\text{base} \mid D.\text{offset} \mid D.\text{asInt}() \\
& \mid & D.\text{write}(d, d); d \mid w.\text{write}(D, d); d \mid w.\text{write}(w, D); d \mid w.\text{write}(w, w); D \\
& \mid & D + d \mid w + D \mid \text{let } y = D \text{ in } d \mid \text{let } y = w \text{ in } D
\end{array}
$$

$$
\frac{l \notin \text{dom}(I) \qquad n > 0}{I, \text{A}; \text{new FatBlock}(n) \rightarrow I\{l := \text{FatBlock}(\text{A}, \text{new FatPtr}(\text{null}, 0) \times n)\}, \text{A} + n; l} \quad \text{(Java-FatBlock)}
$$

$$
\frac{I(l) = \text{FatBlock}(\text{a}, [w_0, \ldots, w_n, \ldots, w_m])}{I, \text{A}; l.\text{read}(n) \rightarrow I, \text{A}; w_n} \quad \text{(Java-read)}
$$

$$
\frac{I(l) = \text{FatBlock}(\text{a}, [w_0, \ldots, w_n, \ldots, w_m])}{I, \text{A}; l.\text{write}(n, w); d \rightarrow I\{l := \text{FatBlock}(\text{a}, [w_0, \ldots, w, \ldots, w_m])\}, \text{A}; d} \quad \text{(Java-write)}
$$

$$
I, \text{A}; \text{new FatPtr}(l, n).\text{base} \rightarrow I, \text{A}; l \quad \text{(Java-base)}
$$

$$
I, \text{A}; \text{new FatPtr}(\text{null}, n).\text{base} \rightarrow I, \text{A}; \text{null} \quad \text{(Java-base-null)}
$$

$$
I, \text{A}; \text{new FatPtr}(l, n).\text{offset} \rightarrow I, \text{A}; n \quad \text{(Java-offset)}
$$

$$
I, \text{A}; \text{new FatPtr}(\text{null}, n).\text{offset} \rightarrow I, \text{A}; n \quad \text{(Java-offset-null)}
$$

$$
\frac{I(l) = \text{FatBlock}(\text{a}, [w_0, \ldots, w_m])}{I, \text{A}; \text{new FatPtr}(l, n).\text{asInt}() \rightarrow I, \text{A}; \text{a} + n} \quad \text{(Java-asInt)}
$$

$$
I, \text{A}; \text{new FatPtr}(\text{null}, n).\text{asInt}() \rightarrow I, \text{A}; n \quad \text{(Java-asInt-null)}
$$

$$
I, \text{A}; \text{let } y = w \text{ in } d \rightarrow I, \text{A}; d[w/y] \quad \text{(Java-let)}
$$

$$
\frac{I, \text{A}; d \rightarrow I', \text{A}'; d'}{I, \text{A}; D[d] \rightarrow I', \text{A}'; D[d']} \quad \text{(Java-context)}
$$

図 8 Java サブセットの簡約規則

(Java-asInt-null) より，

$$
\begin{array}{rl}
& [\![H]\!]_\epsilon, \text{addr}(H); \\
& [\![\text{ptr}(b_1, \mathit{off}_1) \oplus \text{num}(n_2)]\!]_\epsilon \\
= & [\![H]\!]_\epsilon, \text{addr}(H); \\
& \text{let } y = \text{new FatPtr}(\epsilon(b_1), \mathit{off}_1) \text{ in} \\
& \text{new FatPtr}(y.\text{base}, y.\text{offset} + \\
& \quad \text{new FatPtr}(\text{null}, n_2).\text{asInt}()) \\
\rightarrow^* & [\![H]\!]_\epsilon, \text{addr}(H); \\
& \text{new FatPtr}(\epsilon(b_1), \mathit{off}_1 + n_2) \\
= & [\![H]\!]_\epsilon, \text{addr}(H); [\![\text{ptr}(b_1, \mathit{off}_1 + n_2)]\!]_\epsilon
\end{array}
$$

したがって，$\epsilon' = \epsilon$ と取れば良い．

(C-ptr-add2)，(C-ptr-add3)，および (C-ptr-add4) の場合は，(C-ptr-add1) の場合と同様である．

最後に使われた規則が (C-alloc-num) の場合，$b = \text{addr}(H)$，$n > 0$，$H' = H\{b := \text{num}(0) \times n\}$，$e = \text{malloc}(\text{num}(n))$，$e' = \text{ptr}(b, 0)$ とする．$\epsilon' = \epsilon \cup \{(b, l)\}$ と取れば，(trans-heap) と (trans-num) より，

$$
\begin{array}{l}
[\![H']\!]_{\epsilon'}(\epsilon'(b)) = \\
\qquad \text{FatBlock}(b, \text{new FatPtr}(\text{null}, 0) \times n) \\
\text{addr}(H') = \text{addr}(H) + n
\end{array}
$$

また，(trans-alloc)，(trans-num) および (Java-asInt-

null)，(Java-FatBlock) より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![\mathrm{malloc}(\mathrm{num}(n))]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \mathrm{new\ FatPtr}(\mathrm{new\ FatBlock}(\\
& \quad \mathrm{new\ FatPtr}(\mathrm{null}, n).\mathrm{asInt}()), 0) \\
\to^* \quad & [\![H']\!]_{\epsilon'}, \mathrm{addr}(H'); \mathrm{new\ FatPtr}(l, 0) \\
= \quad & [\![H']\!]_{\epsilon'}, \mathrm{addr}(H'); [\![\mathrm{ptr}(b, 0)]\!]_{\epsilon'}
\end{aligned}$$

(C-alloc-ptr) の場合は，(C-alloc-num) の場合と同様である．

最後に使われた規則が (C-let) の場合，$H = H'$，e は $\mathrm{let}\ x = v\ \mathrm{in}\ e_0$，$e' = e_0[v/x]$ とする．(trans-let)，(Java-let) および補題 3 より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![\mathrm{let}\ x = v\ \mathrm{in}\ e_0]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \mathrm{let}\ x = [\![v]\!]_\epsilon\ \mathrm{in}\ [\![e_0]\!]_\epsilon \\
\to^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![e_0]\!]_\epsilon[[\![v]\!]_\epsilon/x] \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![e_0[v/x]]\!]_\epsilon
\end{aligned}$$

したがって，$\epsilon' = \epsilon$ と取れば良い．

最後に使われた規則が (C-context) の場合，$e = C[e_0]$，$e' = C[e_0']$，$H; e_0 \to H'; e_0'$ とする．帰納法の仮定より，

$\forall \epsilon. \exists \epsilon'.$

$[\![H]\!]_\epsilon, \mathrm{addr}(H); [\![e_0]\!]_\epsilon \to^* [\![H']\!]_{\epsilon'}, \mathrm{addr}(H'); [\![e_0']\!]_{\epsilon'}$

したがって，補題 1 より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![C[e_0]]\!]_\epsilon \\
& \to^* [\![H']\!]_{\epsilon'}, \mathrm{addr}(H'); [\![C[e_0']]\!]_{\epsilon'}
\end{aligned}$$

次に定理 2 を，式 e の構文に関する帰納法で証明する．

$e = x$ の場合は明らか．

$e = *e_1$ の場合，$H; e \not\to$ より $H; e_1 \not\to$．e_1 が値でない場合は補題 2 より，

$\forall \epsilon.\ \exists I, \mathrm{A}, d.\ [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![*e_1]\!]_\epsilon \to^* I, \mathrm{A}; d \not\to$ かつ d は値でない．したがって，e_1 が値の場合を考えれば十分である．$e_1 = \mathrm{num}(n)$ の場合，(trans-deref)，(trans-num) および (Java-let)，(Java-base-null)，(Java-offset-null) より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![*(\mathrm{num}(n))]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \mathrm{let}\ y = \mathrm{new\ FatPtr}(\mathrm{null}, n)\ \mathrm{in} \\
& y.\mathrm{base.read}(y.\mathrm{offset}) \\
\to^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \mathrm{null.read}(n) \\
\not\to &
\end{aligned}$$

したがって，$I = [\![H]\!]_\epsilon$，$\mathrm{A} = \mathrm{addr}(H)$，$d = \mathrm{null.read}(n)$ と取れば良い．$e_1 = \mathrm{ptr}(b, \mathit{off})$ かつ $b \notin \mathrm{dom}(H)$ の場合，(trans-heap) より $\epsilon(b) \notin \mathrm{dom}([\![H]\!]_\epsilon)$．また，(trans-deref)，(trans-ptr) および (Java-let)，(Java-base)，(Java-offset) より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![*(\mathrm{ptr}(b, \mathit{off}))]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \mathrm{let}\ y = \mathrm{new\ FatPtr}(\epsilon(b), \mathit{off})\ \mathrm{in} \\
& y.\mathrm{base.read}(y.\mathrm{offset}) \\
\to^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \epsilon(b).\mathrm{read}(\mathit{off}) \\
\not\to \quad & (\because \epsilon(b) \notin \mathrm{dom}([\![H]\!]_\epsilon))
\end{aligned}$$

したがって，$I = [\![H]\!]_\epsilon$，$\mathrm{A} = \mathrm{addr}(H)$，$d = \epsilon(b).\mathrm{read}(\mathit{off})$ と取れば良い．$e_1 = \mathrm{ptr}(b, \mathit{off})$ で $H(b) = [v_0, \ldots, v_m]$ かつ $\mathit{off} < 0 \vee \mathit{off} > m$ の場合，(trans-heap) より，

$$[\![H]\!]_\epsilon(\epsilon(b)) = \mathrm{FatBlock}(b, [[\![v_0]\!]_\epsilon, \ldots . [\![v_m]\!]_\epsilon]);$$

また，(trans-deref)，(trans-ptr) および (Java-let)，(Java-base)，(Java-offset) より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![*(\mathrm{ptr}(b, \mathit{off}))]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \mathrm{let}\ y = \mathrm{new\ FatPtr}(\epsilon(b), \mathit{off})\ \mathrm{in} \\
& y.\mathrm{base.read}(y.\mathrm{offset}) \\
\to^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \epsilon(b).\mathrm{read}(\mathit{off}) \\
\not\to \quad & (\because \mathit{off} < 0 \vee \mathit{off} > m)
\end{aligned}$$

したがって，$I = [\![H]\!]_\epsilon$，$\mathrm{A} = \mathrm{addr}(H)$，$d = \epsilon(b).\mathrm{read}(\mathit{off})$ と取れば良い．

e が $*e_1 = e_2; e_3$ の場合は，$e = *e_1$ の場合と同様に，e_1 と e_2 が値の場合を考えれば十分である．$e_1 = \mathrm{num}(n)$ の場合，(trans-update)，(trans-num) および (Java-let)，(Java-base-null)，(Java-offset-null) より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![*(\mathrm{num}(n)) = v; e_3]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \mathrm{let}\ y = \mathrm{new\ FatPtr}(\mathrm{null}, n)\ \mathrm{in} \\
& y.\mathrm{base.write}(y.\mathrm{offset}, [\![v]\!]_\epsilon); [\![e_3]\!]_\epsilon \\
\to^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \mathrm{null.write}(n, [\![v]\!]_\epsilon); [\![e_3]\!]_\epsilon \\
\not\to &
\end{aligned}$$

したがって，$I = [\![H]\!]_\epsilon$，$\mathrm{A} = \mathrm{addr}(H)$，$d = \mathrm{null.write}(n, [\![v]\!]_\epsilon); [\![e_3]\!]_\epsilon$ と取れば良い．$e_1 = \mathrm{ptr}(b, \mathit{off})$ かつ $b \notin \mathrm{dom}(H)$ の場合，(trans-heap) より $\epsilon(b) \notin \mathrm{dom}([\![H]\!]_\epsilon)$．また，(trans-update)，(trans-ptr) および (Java-let)，(Java-base)，(Java-offset)

より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![*(\mathrm{ptr}(b, \mathit{off})) = v; e_3]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \text{let } y = \text{new FatPtr}(\epsilon(b), \mathit{off}) \text{ in} \\
& y.\text{base.write}(y.\text{offset}, [\![v]\!]_\epsilon); [\![e_3]\!]_\epsilon \\
\to^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \epsilon(b).\text{write}(\mathit{off}, [\![v]\!]_\epsilon); [\![e_3]\!]_\epsilon \\
\not\to \quad & (\because \epsilon(b) \notin \mathrm{dom}([\![H]\!]_\epsilon))
\end{aligned}$$

したがって，$I = [\![H]\!]_\epsilon$，$\mathrm{A} = \mathrm{addr}(H)$，$d = \epsilon(b).\text{write}(\mathit{off}, [\![v]\!]_\epsilon); [\![e_3]\!]_\epsilon$ と取れば良い．$e_1 = \mathrm{ptr}(b, \mathit{off})$ で $H(b) = [v_0, \ldots, v_m]$ かつ $\mathit{off} < 0 \vee \mathit{off} > m$ の場合，(trans-heap) より，

$$[\![H]\!]_\epsilon(\epsilon(b)) = \mathrm{FatBlock}(b, [[\![v_0]\!]_\epsilon, \ldots . [\![v_m]\!]_\epsilon]);$$

また，(trans-update)，(trans-ptr) および (Java-let)，(Java-base)，(Java-offset) より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![*(\mathrm{ptr}(b, \mathit{off})) = v; e_3]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \text{let } y = \text{new FatPtr}(\epsilon(b), \mathit{off}) \text{ in} \\
& y.\text{base.write}(y.\text{offset}, [\![v]\!]_\epsilon); [\![e_3]\!]_\epsilon \\
\to^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \epsilon(b).\text{write}(\mathit{off}, [\![v]\!]_\epsilon); [\![e_3]\!]_\epsilon \\
\not\to \quad & (\because \mathit{off} < 0 \vee \mathit{off} > m)
\end{aligned}$$

したがって，$I = [\![H]\!]_\epsilon$，$\mathrm{A} = \mathrm{addr}(H)$，$d = \epsilon(b).\text{write}(\mathit{off}, [\![v]\!]_\epsilon); [\![e_3]\!]_\epsilon$ と取れば良い．

$e = (t)e_1$ かつ e_1 が値の場合，$H; (t)e_1 \not\to$ となるような e_1 は存在しない．

$e = e_1 + e_2$ かつ e_1 と e_2 が値の場合を考える．$e_1 = \mathrm{ptr}(b_1, \mathit{off}_1)$，$e_2 = \mathrm{num}(n_2)$ かつ $b_1 \notin \mathrm{dom}(H)$ の場合，(trans-heap) より $\epsilon(b_1) \notin \mathrm{dom}([\![H]\!]_\epsilon)$．また，(trans-int-add)，(trans-ptr)，(trans-num) より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![\mathrm{ptr}(b_1, \mathit{off}_1) + \mathrm{num}(n_2)]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \text{new FatPtr(null,} \\
& \qquad \text{new FatPtr}(\epsilon(b_1), \mathit{off}_1).\text{asInt}() + \\
& \qquad\quad \text{new FatPtr(null}, n_2).\text{asInt}()) \\
\not\to \quad & (\because \epsilon(b_1) \notin \mathrm{dom}([\![H]\!]_\epsilon))
\end{aligned}$$

したがって，$I = [\![H]\!]_\epsilon$，$\mathrm{A} = \mathrm{addr}(H)$，$d = \text{new FatPtr(null, new FatPtr}(\epsilon(b_1), \mathit{off}_1).\text{asInt}() + \text{new FatPtr(null}, n_2).\text{asInt}())$ と取れば良い．$e_1 = \mathrm{num}(n_1)$，$e_2 = \mathrm{ptr}(b_2, \mathit{off}_2)$ かつ $b_2 \notin \mathrm{dom}(H)$ の場合，および $e_1 = \mathrm{ptr}(b_1, \mathit{off}_1)$，$e_2 = \mathrm{ptr}(b_2, \mathit{off}_2)$ かつ $\{b_1, b_2\} \not\subseteq \mathrm{dom}(H)$ の場合も同様である．

$e = e_1 \oplus e_2$ の場合は，$e = e_1 + e_2$ の場合と同様である．

$e = \mathrm{malloc}(e_1)$ かつ e_1 が値の場合を考える．$e_1 = \mathrm{num}(n)$ かつ $n \leq 0$ の場合，(trans-alloc)，(trans-num) および (Java-asInt-null) より，

$$\begin{aligned}
& [\![H]\!]_\epsilon, \mathrm{addr}(H); [\![\mathrm{malloc}(e_1)]\!]_\epsilon \\
= \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \text{new FatPtr(new FatBlock(} \\
& \qquad \text{new FatPtr(null}, n).\text{asInt}()), 0) \\
\to^* \quad & [\![H]\!]_\epsilon, \mathrm{addr}(H); \\
& \text{new FatPtr(new FatBlock}(n), 0) \\
\not\to \quad & (\because n \leq 0)
\end{aligned}$$

したがって，$I = [\![H]\!]_\epsilon$，$\mathrm{A} = \mathrm{addr}(H)$，$d = \text{new FatPtr(new FatBlock}(n), 0)$ と取れば良い．$e_1 = \mathrm{ptr}(b, \mathit{off})$ かつ $b \notin \mathrm{dom}(H)$ の場合は，$e = e_1 + e_2$ の場合と同様である．

e が let $x = e_1$ in e_2 かつ e_1 と e_2 が値の場合，$H; \text{let } x = e_1 \text{ in } e_2 \not\to$ となるような e_1，e_2 は存在しない．

B 未定義動作の扱い

1 節で述べた変換の方針が実現可能か調べるために，我々は C 言語の仕様 [1] において，「未定義」と明示されている動作をすべて抽出した．さらに，それらの動作の具体的な実装法を検討し，我々の方針で変換が可能であることを確認した．(ただし，ライブラリの仕様である第 7 章を除く．ライブラリへの対応は今後の検討課題である．また，C 言語の仕様には「未定義」と明示されていない未定義動作も存在するが，それらは網羅していない．)

ポインタ演算に関連する主な未定義動作について以下に述べる．

6.3.2.3 節（35 頁）「任意のポインタ型は整数型に型変換できる．…結果が整数型で表現できなければ，その動作は未定義とする．」十分に大きな整数とみなしてから切り捨てる，という安全な動作で置き換える．

6.3.2.3 節（35 頁）「オブジェクト型又は不完全型へのポインタは，他のオブジェクト型又は不完全型へのポインタに型変換できる．その結果のポインタが，被参照型に関して正しく境界調整されていなければ，その動作は未定義とする．」 4 節，5 節で述べたよう

に，ポインタの型変換自体は，何もしないという動作に置き換え，エラーとはしない．ただし，正しく境界調整されていないポインタを介して，実際にオブジェクトにアクセスした場合は実行時エラーとする．

6.5.3.2 節（58 頁）「正しくない値がポインタに代入されている場合，単項*演算子の動作は，未定義とする．」変換された Java プログラムにおいて実行時エラーとする．

6.5.6 節（61 頁）「整数型をもつ式をポインタに加算又はポインタから減算する場合，…それ以外の場合，動作は未定義とする．」ポインタと整数の加算又は減算自体は，エラーとはしない．ただし，演算前のポインタがある配列内を指し，演算後のポインタが同じ配列内を指していない場合，演算後のポインタを介したアクセスは実行時エラーとする．

例として，int 型変数 a のアドレスを値に持つ（int へのポインタ型）変数 p に 1 を加算した式を，単項* 演算子のオペランドとする C コードを考える．この* 演算子の動作は未定義であり，C 言語処理系として一般的な GCC では，不定の値が返される場合が多い．それに対し我々の処理系では，実行時エラーが発生する．

6.5.8 節（63 頁）「二つのポインタを比較する場合，その結果は指されているオブジェクトのアドレス空間内の相対位置に依存する．…その他のすべての場合，動作は未定義とする．」ポインタを整数にキャストして比較するという安全な動作で置き換える．ただし 3 節で述べたように，現在の実装では，AddrCounter がオーバーフローした場合，ポインタを整数にキャストして比較した結果が矛盾することがありうる．

なお，仕様 3.4 節（3 頁）で定義されているように，「動作」には以下の 4 つが含まれる．

未規定の動作 「この規格が，ニつ以上の可能性を提供し，個々の場合にどの可能性を選択するかに関して何ら要求を課さない動作」（例：関数の実引数の評価順序）

処理系定義の動作 「未規定の動作のうち，各処理系が選択した動作を文書化するもの」（例：符号付き整数を右シフトした場合の最上位ビットの伝播法）

文化圏固有動作 「国家，文化及び言語の地域規約に依存する動作であり，各処理系がその動作を文書化するもの」（例：islower 関数が 26 種類のラテン小文字以外の文字に対し，真を返すかどうか）

未定義の動作 「可搬性がない若しくは正しくないプログラム構成要素を使用したときの動作，又は正しくないデータを使用したときの動作であり，この規格が何ら要求を課さないもの」（例：整数演算のオーバーフローに対する動作）

我々の実装も一つの処理系であるので，未定義以外の 3 つの動作に対しては，エラーとせず特定の動作を与える（本研究は移植性の問題を対象とするものではなく，処理系ごとの差異については関知しない）．したがって，「エラーとするか，安全な動作で置き換えるか」という検討が必要となるのは，未定義の動作のみである．