Java言語への変換による
ポインタ演算の安全な実装方式

または：
C言語とは何か？

上嶋 祐紀
元 東北大学
（現 東芝）

住井 英二郎
東北大学

背景

- 「C言語は危険」
- 「C言語は低水準」

（よりによって）
http://ja.wikipedia.org/wiki/C言語

特徴 [編集]

Cは手続き型言語であり、コンパイラ言語として設計された。Cは、自由度、実行速度、コンパイル速度などを追求したが、代わりにコンパイル後のコードの安全性を犠牲にもしているので、コンピュータ寄りの言語仕様になっている。

- アマチュアから
  い。Cは正負の
  しているが、最
  れている。
- パソコンはもち
  まで、Cを使用
  派生言語も多
- 採用されてい
  程度の言語適
  祥元であるUN

野の棲み分けができていた面があったのだが、幅広い領域に移植された結果、適切でない分野にCが使われている場面もある。

- 低水準な記述が出来る高級言語とも、高級言語の顔をした低級言語とも言われる。また、コーディング上の“自由度”が非常に高い。そのため良くも悪くも“何でも出来てしまう”パワフルさは多くのプログラマの支持を集める一方で、セキュリティー脆弱性や潜んだバグによる想定外の動作、コンパイラによる最適化の難しさ（そのためCはコンパイラ言語として決して高速ではない）といった欠点の原因ともなっている。
- 商用・非商用を問わずコンパイラやC向けのエディタが豊富で開発

背景

- 「C言語は危険」
- 「C言語は低水準」

本当？

JIS

プログラム言語 C

JIS X 3010 : 2003

(ISO/IEC 9899 : 1999)

(ITSCJ/JSA)

平成 15 年 12 月 20 日 改正

日本工業標準調査会 審議

(日本規格協会 発行)

[] 演算子が暗黙に意味する単項*演算子は評価されず，&演算子を削除し [] 演算子を+演算子に変更した場合と同じ結果となる。これら以外の場合，結果はそのオペランドが指し示すオブジェクト又は関数へのポインタとなる。

単項*演算子は，間接参照を表す。オペランドが関数を指している場合，その結果は関数指示子とする。オペランドがオブジェクトを指している場合，その結果はそのオブジェクトを指し示す左辺値とする。オペランドが型“～型へのポインタ”をもつ場合，その結果は型“～型”をもつ。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(83)。

前方参照　記憶域クラス指定子（6.7.1），構造体指定子及び共用体指定子（6.7.2.1）

6.5.3.3　単項算術演算子

制約　単項+演算子及び単項-演算子のオペランドは，算術型をもたなければならない。~演算子のオペランドは，整数型をもたなければならない。!演算子のオペランドは，スカラ型をもたなければならない。

意味規則　単項+演算子の結果は，その（拡張された）オペランドの値とする。オペランドに対して整数拡張を行い，その結果は，拡張された型をもつ。

単項-演算子の結果は，その（拡張された）オペランドの符号を反転した値とする。オペランドに対して整数拡張を行い，その結果は，拡張された型をもつ。

~演算子の結果は，その（拡張された）オペランドのビット単位の補数とする（すなわち，結果の各ビットは，拡張されたオペランドの対応するビットがセットされていない場合，そしてその場合に限り，セットされる。）。オペランドに対して整数拡張を行い，その結果は，拡張された型をもつ。拡張された型が符号無し整数型である場合，~E はその型で表現可能な最大値から E を減算した値と等価とする。

— 左オペランドがオブジェクト型へのポインタで，右オペランドの型が整数型である。(減分は 1 の減算に等しい。)

意味規則 両オペランドが算術型をもつ場合，通常の算術型変換をそれらに適用する。

2 項+演算子の結果は，両オペランドの和とする。

2 項-演算子の結果は，第 1 オペランドから第 2 オペランドを引いた結果の差とする。

これらの演算子に関しては，配列の要素でないオブジェクトへのポインタは，要素型としてそのオブジェクトの型をもつ長さ 1 の配列の最初の要素へのポインタと同じ動作をする。

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，結果は，ポインタオペランドの型をもつ。ポインタオペランドが配列オブジェクトの要素を指し，配列が十分に大きい場合，その結果は，その配列の要素を指し，演算結果の要素と元の配列要素の添字の差は，整数式の値に等しい。すなわち，式 P が配列オブジェクトの *i* 番目の要素を指している場合，式 (P)+N (N+(P) と等しい) 及び (P)-N (N は値 *n* をもつと仮定する。) は，それらが存在するのであれば，それぞれ配列オブジェクトの *i*+*n* 番目及び *i*−*n* 番目の要素を指す。さらに，式 P が配列オブジェクトの最後の要素を指す場合，式 (P)+1 はその配列オブジェクトの最後の要素を一つ越えたところを指し，式 Q が配列オブジェクトの最後の要素を一つ越えたところを指す場合，式 (Q)-1 はその配列オブジェクトの最後の要素を指す。ポインタオペランド及びその結果の両方が同じ配列オブジェクトの要素，又は配列オブジェクトの最後の要素を一つ越えたところを指している場合，演算によって，オーバフローを生じてはならない。それ以外の場合，動作は未定義とする。結果が配列オブジェクトの最後の要素を一つ越えたところを指す場合，評価される単項*演算子のオペランドとしてはならない。

Strict Aliasing Rules

```
> cat alias.c
main() {
  int i = 0x12345678;
  short *p = &i;
  printf("%x %x¥n", *p, *(p+1));
}
> gcc -O alias.c ; ./a.out
5678 1234
> gcc -O2 alias.c ; ./a.out
5678 4002
```

いか，又は結果の型で表現可能な値の範囲にない場合），その動作は未定義とする。

格納された値にアクセスするときのオブジェクトの有効型（effective type）は，（もしあれば）そのオブジェクトの宣言された型とする(71)。宣言された型をもたないオブジェクトへ，文字型以外の型をもつ左辺値を通じて値を格納した場合，左辺値の型をそのアクセス及び格納された値を変更しないそれ以降のアクセスでのオブジェクトの有効型とする。宣言された型をもたないオブジェクトに，memcpy 関数若しくは memmove 関数を用いて値をコピーするか，又は文字型の配列として値をコピーした場合，そのアクセス及び値を変更しないそれ以降のアクセスでのオブジェクトの有効型は，値のコピー元となったオブジェクトの有効型があれば，その型とする。宣言された型をもたないオブジェクトに対するその他のすべてのアクセスでは，そのアクセスでの左辺値の型を有効型とする。

オブジェクトに格納された値に対するアクセスは，次のうちのいずれか一つの型をもつ左辺値によらなければならない(77)。

- オブジェクトの有効型と適合する型
- オブジェクトの有効型と適合する型の修飾版
- オブジェクトの有効型に対応する符号付き型又は符号無し型
- オブジェクトの有効型の修飾版に対応する符号付き型又は符号無し型
- メンバの中に上に列挙した型の一つを含む集成体型又は共用体型（再帰的に包含されている部分集成体又は含まれる共用体のメンバを含む。）
- 文字型

浮動小数点型の式は短縮（contract）してもよい。すなわち，ハードウェアによる不可分な操作として評

---

(71) 構文は，式の評価における演算子の優先順位を指定する。それは 6.5 の箇条の順序と同じであり，最初の箇条が最も高い優先順位をもつ。したがって，例えば 2 項+演算子（6.5.6）のオペランドとして許される式は，6.5.1～6.5.6 で定義された式である。例外は，オペランドとしてキャスト式（6.5.4）

http://mail-index.netbsd.org/tech-kern/2003/08/11/0001.html

(linked from http://gcc.gnu.org/bugs/#nonbugs_c)

"ISO C is not your grandfather's C, and it is wrong to think of it as a high-level machine language..."

（色付話者）

「未定義」と「危険」の関係

- 未定義　⊃　危険
  - 例：バッファオーバーフロー

- 未定義　≠　危険
  - ~~プログラムがバッファオーバーフローしたら不正侵入を許さなければならない？~~
  - プログラムがバッファオーバーフローしたら実行を中止しても良い

Fail-Safe C
[大岩,住井,米澤'01]

- 「未定義」とされる動作をすべて「実行の中止」または「特定の動作」で「定義」したC言語の実装
  - [Oiwa PLDI'09]でC89を完全に実装
  - OpenSSL, OpenSSH, ISC BIND, thttpd, qmail, postfix, libtiff, zlib などが動作
  - オーバーヘッドは1～8倍程度

    Cf. CCured [Necula et al. POPL'02]

FAQ

[Q] C言語では（Schemeなどでも）
関数引数の評価順序なども「不定」だが
「危険」なのか？

[A] C言語の標準規格では（動作が）
「不定」という用語は存在しない。

Cf. 値が「不定」(indeterminate)＝未規定∨トラップ表現

関数引数の評価順序は「未定義」ではなく
「未規定」なので「危険」とみなさない。

JIS:「3.4.4 未規定の動作（unspecified behavior）
この規格が，二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。」

Fail-Safe C to Java [上嶋,住井'07]

- 「危険」な(とされている)言語を「安全」な(とされている)言語にコンパイル
- もしコンパイラにバグがあっても、コンパイルしたプログラムは「安全」
- 「低水準」言語から「高水準」言語へのコンパイル(通常と逆向き！)なのでより困難
  - オーバーヘッドはGCC4の1.3～6倍程度、手書きJavaの1～6.6倍程度

「安全」(safe)とは何か？

A Syntactic Approach to Type Soundness

Andrew K. Wright* Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

June 18, 1992

Rice Technical Report TR91-160
To appear in: *Information and Computation*

「安全」(safe)とは何か？

[Wright-Felleisen'92]

- プログラムの(操作的)意味論を
  ーステップごとの状態遷移「→」で定義
- プログラム状態Pに対し、**P→QなるQが存在せず、**Pが終了状態でもないとき、「Pはstuck」と定義
  - まさに「動作が未定義」
  - 我々は「**stuck状態にならなければ安全(safe)**」と定義
    - "safe" ≠ "correct"

実装の概要1: 配列

- C言語のメモリ　＝　配列
  - 「正しい型」(有効型)の値の列
  - 単なるフラットな「メモリ」ではない
    - Recall the aliasing rules:
      正しい型とchar*以外でのアクセスは未定義

- C言語の変数　＝　要素数1の配列

実装の概要1: 配列

```
abstract class Block {
  int objsize; // 要素一個のサイズ
  int size; // 配列全体のサイズ
  int addr; // 先頭仮想アドレス
  abstract byte readbyte(int vo);
  abstract void
    writebyte(int vo, byte b);
  abstract double readdouble(int vo);
  abstract double
    writedouble(int vo, double d);
  ... }
```

アクセス
メソッド

仮想オフセット

実装の概要1: 配列

```
class doubleBlock extends Block {
  private double[] contents;
  doubleBlock(int n) {
    objsize = 8; size = 8*n;
    addr = 仮想先頭アドレスを新規生成;
    contents = new double[n]; }
  double readdouble(int vo) {
    if (vo%8 != 0) エラー;
    return contents[vo / 8]; }
  // readbyte等も頑張ってエミュレート
  ... }
```

実装の概要2: ポインタ

誤：

ポインタ　＝　アドレス

正：

ポインタ　＝

配列の要素への参照

実装の概要2: Fatポインタ

```
class FatPtr {
  Block base; int offset;
  FatPtr(Block b, int vo) {
    base = b; offset = vo; }
  int asint() {
    if (base == NULL) return offset;
    return base.addr + offset; } }
```

- pがT*型のポインタのとき、
  右辺式*pはp.base.readT(P.offset)に、
  代入文*p=xはp.base.writeT(P.offset,x)
  にそれぞれ変換

実装の概要3: Fat整数

7.18.1.4 オブジェクトを指すポインタを保持可能な整数型

定義する型

`intptr_t`

は，次に掲げるすべての特性をもつ符号付き整数型を示す。

- すべての正しい void へのポインタがこの型へ変換可能である。
- この型から void へのポインタに逆変換することが可能であり，その結果が元のポインタと比較して等しくなる。

定義する型

`uintptr_t`

は，次に掲げるすべての特性をもつ符号無し整数型を示す。

- すべての正しい void へのポインタがこの型へ変換可能である。
- この型から void へのポインタに逆変換することが可能であり，その結果が元のポインタと比較して等しくなる。

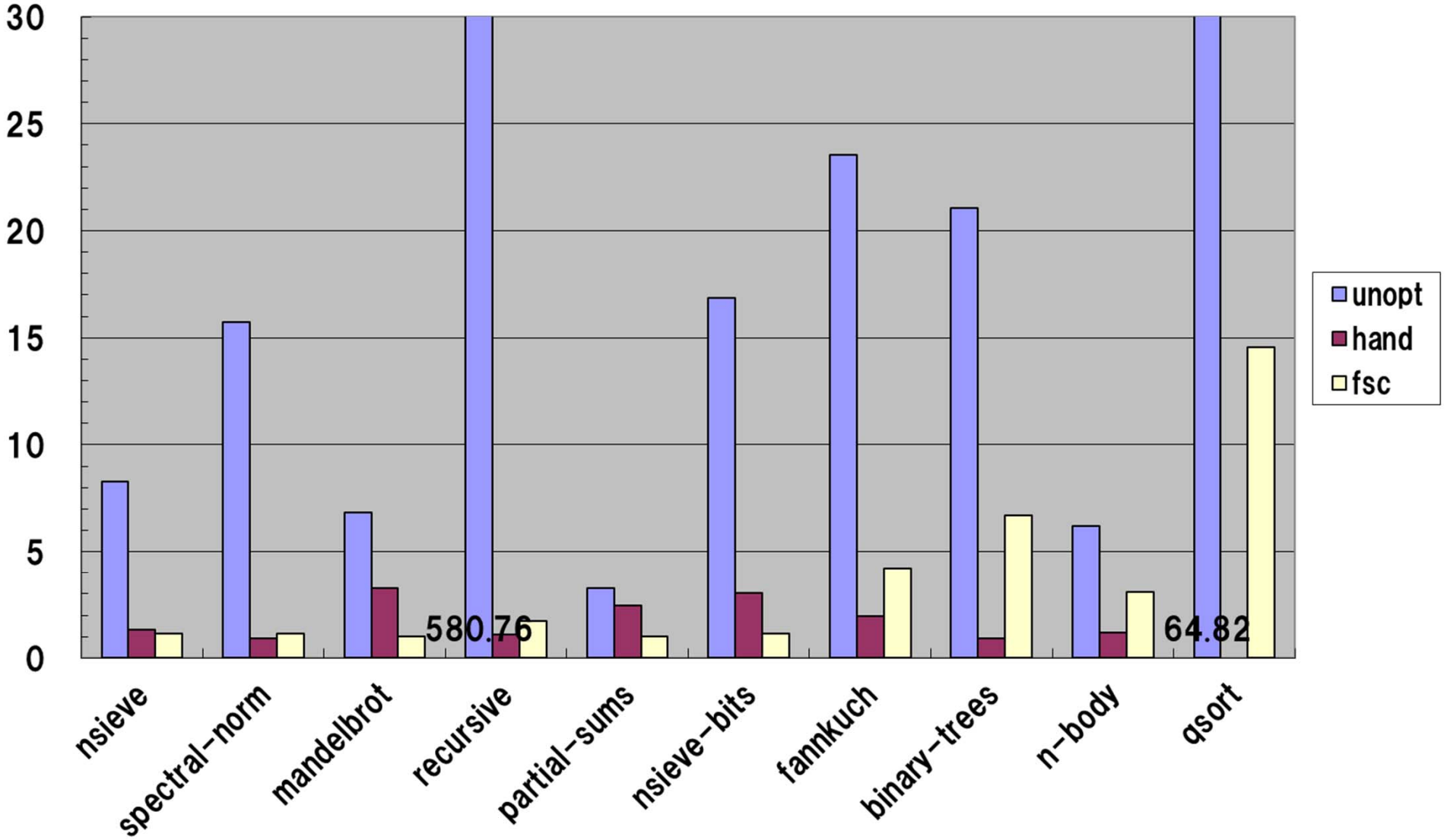
これらの型は，省略可能とする。

実装の概要3: Fat整数

```
class FatInt extends FatPtr {
  Block base; int offset;
  FatInt(int i) {
    base = null; offset = i; }
  int asint() {
    return offset; } }
```

- 実際の実装では、FatIntとFatPtrは親子ではなく、「Fat抽象クラス」を共通の親とする兄弟
  - Sun HotSpot VMでは「なぜか」0～20%速い

実験結果(最適化前)


30
25
20
15
10
5
0
580.76
64.82
unopt
hand
fsc
nsieve
spectral-norm
mandelbrot
recursive
partial-sums
nsieve-bits
fannkuch
binary-trees
n-body
qsort

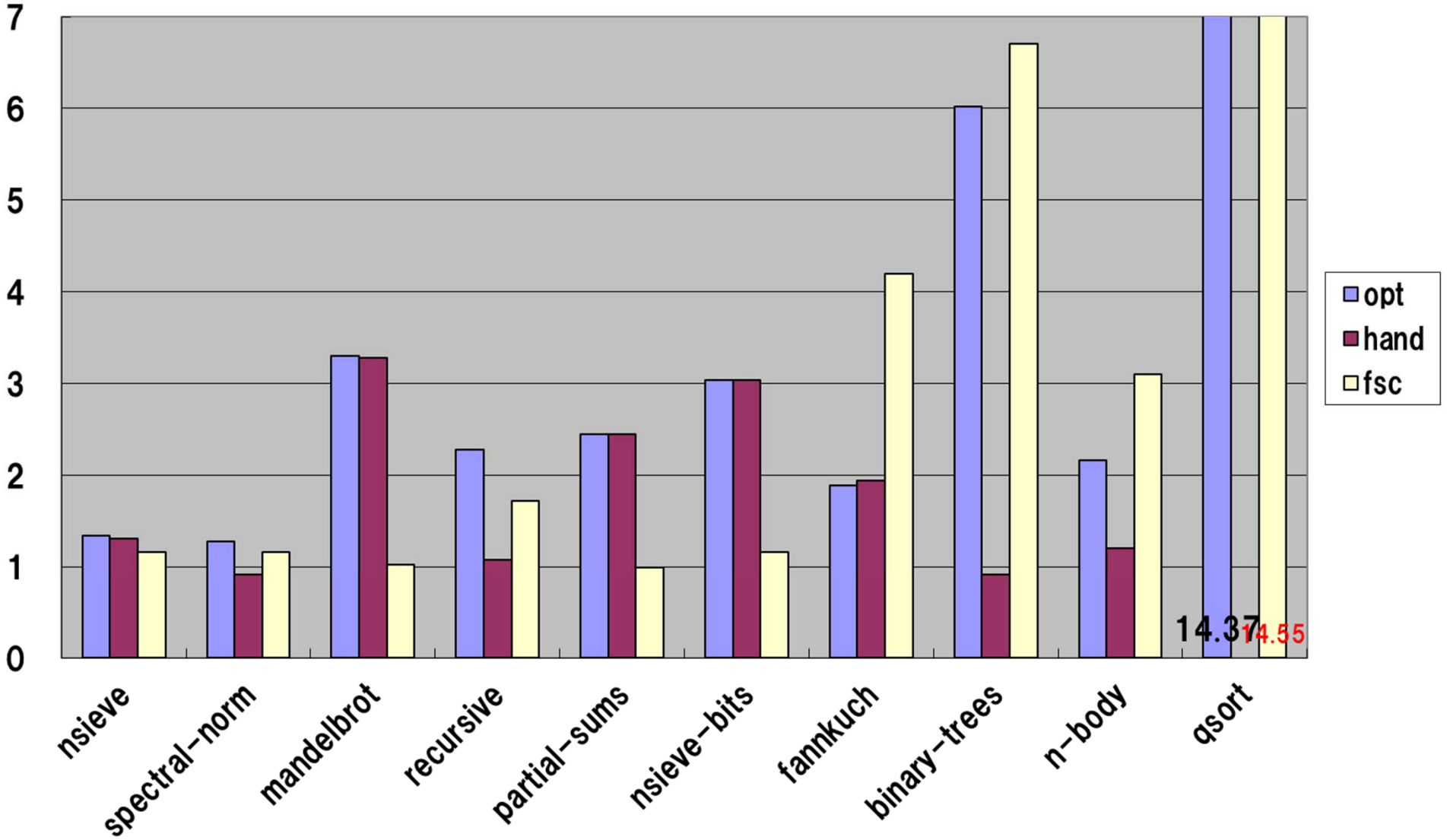
最適化(1/2)

- &をとられない基本型変数
  （および要素の&をとられない配列）は、
  Blockオブジェクトにしない
- ポインタを代入されず、かつ、
  ポインタとして使用されないint型変数
  （およびint型配列の要素）は、
  FatIntオブジェクトにしない
  - 古典的な関数内データフロー解析
- int型の値は、「必要になるまで」
  （＝エスケープするまで）FatIntにしない

最適化(2/2)

- 関数定義f(T x){...}を
  f(T x'){T x = x'; ...}と書き換える
  - x'をBlockにする必要がない
  - 先の条件を満たせばxも
- int型仮引数を持つ関数は
  int版とFat版に複製
  - 文脈依存(多相)解析の簡易実現
  - ポインタ型仮引数を持つ関数も
    FatPtr版とFat版に複製

実験結果(最適化後)


7
6
5
4
3
2
1
0
14.37
14.55
opt
hand
fsc
nsieve
spectral-norm
mandelbrot
recursive
partial-sums
nsieve-bits
fannkuch
binary-trees
n-body
qsort

残るオーバーヘッド

- int型関数戻り値に起因するFatInt

  (recursive, binary-trees)

  ⇒ 関数間解析？

- FatPtrによるポインタアクセス

  (spectral-norm, binary-trees, n-body)

  ⇒ オフセットが常に0のFatPtrを静的検出して（Javaの）単なる参照にする？

A Hindsight

intptr_t = long long > long >= int
とすれば、もっと楽だったかもしれない。

定式化: Cサブセットの抽象構文

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

定式化: Javaサブセットの抽象構文

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

定式化: 変換規則

$$
\begin{array}{rcl}
[\![\mathrm{num}(n)]\!]_\epsilon & = & \text{new FatPtr}(\text{null}, n) \\
[\![\mathrm{ptr}(b, \mathit{off})]\!]_\epsilon & = & \text{new FatPtr}(\epsilon(b), \mathit{off}) \\
[\![x]\!]_\epsilon & = & x \\
[\![*e]\!]_\epsilon & = & \text{let } y = [\![e]\!]_\epsilon \text{ in } y.\text{base.read}(y.\text{offset}) \\
[\![*e_1 = e_2; e_3]\!]_\epsilon & = & \text{let } y = [\![e_1]\!]_\epsilon \text{ in } y.\text{base.write}(y.\text{offset}, [\![e_2]\!]_\epsilon); [\![e_3]\!]_\epsilon \\
[\![(t)\, e]\!]_\epsilon & = & [\![e]\!]_\epsilon \\
[\![e_1 + e_2]\!]_\epsilon & = & \text{new FatPtr}(\text{null}, [\![e_1]\!]_\epsilon.\text{asInt}() + [\![e_2]\!]_\epsilon.\text{asInt}()) \\
[\![e_1 \oplus e_2]\!]_\epsilon & = & \text{let } y = [\![e_1]\!]_\epsilon \text{ in new FatPtr}(y.\text{base}, y.\text{offset} + [\![e_2]\!]_\epsilon.\text{asInt}()) \\
[\![\mathrm{malloc}(e)]\!]_\epsilon & = & \text{new FatPtr}(\text{new FatBlock}([\![e]\!]_\epsilon.\text{asInt}()), 0) \\
[\![\text{let } x = e_1 \text{ in } e_2]\!]_\epsilon & = & \text{let } x = [\![e_1]\!]_\epsilon \text{ in } [\![e_2]\!]_\epsilon
\end{array}
$$

定式化: 定理(概要)

CサブセットとJavaサブセットそれぞれの簡約関係「→」を定義すると、

- e→e' ならば [[e]] →* [[e']]
  - 前者の導出に関する帰納法
- eが値ならば[[e]]も値
  - 定義より自明
- eがstuck状態ならば、[[e]]も有限回でstuck状態に簡約される
  - 実際のJavaでは未定義の動作ではなく例外が発生するので安全

課題

- JavaにないC言語の機能
  （符号なし整数、共用体、関数ポインタ、
  多次元配列、goto文）
  - いずれもおそらく実装可能
- Java以外への変換
  （ML, Schemeなど）
- ライブラリ
  - おそらく最も重要かつ困難
  - 値とメタデータを分離する？(cf. CCured)

結論

- 「C言語は危険」？

  ⇒　安全な実装も可能

  （それが「有用」かどうかは用途による）

- 「C言語は低水準」？

  ⇒　標準規格は高水準

  （それで「十分」かどうかは用途による）

宣伝

- 学生の修論だったので現在停止中
- ご興味があればご連絡ください
  - 前述のhindsight等を踏まえて再実装すれば用途によっては十分に実用的になる可能性
  - ただしライブラリ対応にはマンパワーが必要