An Introduction to OSL

金子 知適

TeamGPS

2009 年 3 月 CSA 例会

目次

Disclaimer: OSL は役に立たない

- Bonanza や Crafty を読むべき
  - 将棋やチェスプログラムに馴染んでいない場合
  - 探索 (pruning/cut, 並列化) に興味が有る場合
  - bitboard の将棋盤に興味が有る場合
- OSL を読むのも悪くないかも
  - 利きのある将棋盤
  - GPS 将棋その物に興味がある場合 (i.e., 弱点を探る)
    - 実現確率 →osl::rating::StandardFeatureSet
    - gps_normal の評価関数 →osl::eval::ProgressEval
    - gps_l の評価関数 →osl::eval::experimental::TestEval
- 開発者にも難読!
  - 開発者以外で読んだとうかがったのは今まで二人?

GPS 将棋と OSL

- GPS 将棋: 研究用のプログラム
  - GPS の由来は game programing seminar (東京大学田中哲朗研究室)
  - 2003 年 一次予選 2 勝 5 敗, 2004 年 二次予選 3 勝 6 敗, 2005 年 二次予選 7 勝 2 敗, 決勝 1 勝 6 敗, 2006-2008 年 総合 10-12 位 (強さの伸びがとまる)
- ソースコード公開の経緯
  - 主に研究者の便宜: 当時は Bonanza がなかった!

    研究論文を書くためには実験が必要だが、将棋プログラム全体を 0 から書くことは手間がかかる
  - ライブラリ部分を OSL として分離
  - (とはいえ) 開発者優先の開発
    - 開発方針の変更 →インターフェース変更
    - Microsoft Windows への移植等 なんとかしたいとは…

利用条件等

- OSL (Open Shogi Library)
  - GPS 将棋のほとんどの機能を実装したライブラリ
  - ライセンスはいわゆる新 BSD
- osl-for-csa
  - OSL 由来の CSA 登録ライブラリ
  - CSA 選手権出場に不要な部分を削除
  - ライセンスは新 BSD
- GPS 将棋
  - OSL に追加する形の、対戦用データ、プログラム等
  - ライセンスは GPL

いずれも

- 個人で楽しむ →制限なし
- 再配布 (販売も含む) →各ライセンスで条件あり

入手先 (2009年3月現在)

- osl-for-csa
  - http://www.computer-shogi.org/library/
  - http://gps.tanaka.ecc.u-tokyo.ac.jp/gpsshogi/index.php?osl%2Fosl-for-csa
- OSL/GPSShogi
  - **公式ページ**: http://gps.tanaka.ecc.u-tokyo.ac.jp/gpsshogi/
  - **最新ソースコード**: http://gps.tanaka.ecc.u-tokyo.ac.jp/cgi-bin/viewvc.cgi/ (Download GNU tarball **をクリック**)
  - **バイナリ配布**: **準備中**

主な開発者と開発方針

- **開発者** A:
  - **方針**: **効率の良い実装をする**
    難読になることを厭わない
  - **担当**: **将棋盤指手生成他 (本日一部を紹介)**
- **開発者** B:
  - **最小限の努力で最大限の効果を目指す**
    コピーペーストも厭わない
  - **将棋の経験を生かした評価項目他**
- **開発者** C:
  - **とりあえず動けば良し 後で書き換える**
    (c.f. **明日から勉強しよう**)
  - **探索** (e.g., alphaBeta2.cc, alphaBeta3.cc)、**詰将棋他**

→ **たまに全員で相談してソースコードを整理**

ランダムプレイヤ

ソースコード: osl/sample/random_play.cc

```
NumEffectState state; // 平手初期局面
std::string line;
while (true) {
  MoveVector moves; // ≈ 指手の配列
  LegalMoves::generate(state, moves); // 合法手生成
  if (moves.empty()) 負け
  Move my_move = moves[random()]; // 自分の手
  ApplyMoveOfTurn::doMove(state, my_move); // 盤更新
  std::getline(std::cin, line);
  Move op_move=csa::strToMove(line, state); //相手の手
  ApplyMoveOfTurn::doMove(state, op_move); // 盤更新
}
```

C++の基本文法

- a.b() メンバ関数呼び出し
- a::b() クラス a 内で定義された関数 b の呼び出し
- a::b() namespace a 内で定義された関数 b の呼び出し

- 体系的に C++を学んだ経験がない方
- →プログラミング言語 C++第 3 版 等 この機会に

目次

将棋プログラムの開発の難しさ

将棋プログラムを書くことは難しい:

- **複雑な機能をバグなしで書く必要がある**
  - **多くのバグは目に見えにくい**: e.g., **さり気なく弱くなる**
- **高速に動作させる必要がある**

解:

- **プログラミング言語の機能を活用する →C++の** template
- **信頼できるライブラリを活用する →**e.g., boost
- **現代的な開発方法を用いる**
  - **テストケースを書く →**e.g., cppunit, boost::test, gtest
  - **ペアプログラミング、コードレビュー**
  - **バージョン管理ツールの有効活用**

(1) template を利用したコンパイル時の高速化

ストーリー

1. 将棋プログラムは条件分岐が多い
2. 条件分岐を減らすためにプログラマが頑張ると大変 (→バグの元)
3. コンパイラに任せると簡単

王手生成の例

```
王手生成(turn, state, kingx, kingy) {
    // 歩で王手
    y = (turn == 先手) ? kingy-2 : kingy+2;
    if ((kingx, y)にいるのが歩なら) {
        promote = (turn == 先手) ? y<=4 : y >= 6;
        if (promote) この歩を成る手を生成
        成らない手を生成
    }
... 以下他の駒
}
```

青字の条件分岐をなんとかしたい!

王手生成の例: 改善案(1) 手で展開

```
if (turn == 先手) {          y = kingy-2;
    if ((kingx, y) にいるのが歩なら) {
        if (y<=4) この歩を成る手を生成
        成らない手を生成
    }
} else {
    y = kingy+2;
    if ((kingx, y) にいるのが歩なら) {
        if (y>=6) この歩を成る手を生成
        成らない手を生成
    }
```

条件分岐はなくなったが、ソースコードが 2 倍近くに。

同じことを離れた二ヶ所に書くため見通しが悪い。

王手生成の例: 改善案(2) 表を引く

```
王手生成(turn, state, kingx, kingy) {
    // 歩で王手
    y = kingy+front_king[turn];
    if ((kingx, y)にいるのが歩なら) {
        promote = y*to_black_y[turn][y]<=4;
        if (promote) この歩を成る手を生成
        成らない手を生成
    }
... 以下他の駒
}
```

読みやすいが、表を引くのは案1よりも遅い

王手生成の例: template の利用

関数の引数 turn **を** template **引数** Turn **に変更する**

```
template ⟨Player Turn⟩ 王手生成(state, kingx, kingy) {
    // 歩で王手
    y = (Turn == 先手) ? kingy-2 : kingy+2;
    if ((kingx, y) にいるのが歩なら) {
        promote = (Turn == 先手) ? y<=4 : y >= 6;
        if (promote) この歩を成る手を生成
        成らない手を生成
    }
... 以下他の駒
}
```

王手生成の例: template 関数の呼び出し

- 王手生成⟨先手⟩(state, 4, 6); などと呼び出す
- →Turn に先手が代入されたコードがコンパイルされる

```
// 歩で王手
 y = (Turn == 先手) ? kingy-2 : kingy+2;
 if ((kingx, y) にいるのが歩なら) {
     promote = (Turn == 先手) ? y<=4 : y >= 6;
     if (promote) この歩を成る手を生成
     成らない手を生成
 }
```

王手生成の例: template 関数の呼び出し

- 王手生成〈先手〉(state, 4, 6); などと呼び出す
- →Turn に定数「先手」が代入されたコードがコンパイルされる
- →コンパイラがコードを簡単にする

```
// 歩で王手
 y = (先手 == 先手) ? kingy-2 : kingy+2;
 if ((kingx, y) にいるのが歩なら) {
     promote = (先手 == 先手) ? y<=4 : y >= 6;
     if (promote) この歩を成る手を生成
     成らない手を生成
 }
```

王手生成の例: template 関数の呼び出し

- 王手生成〈先手〉(state, 4, 6); などと呼び出す
- →Turn に定数「先手」が代入されたコードがコンパイルされる
- →コンパイラがコードを簡単にする

```
// 歩で王手
 y = kingy-2;
 if ((kingx, y) にいるのが歩なら) {
     promote = y<=4;
     if (promote) この歩を成る手を生成
     成らない手を生成
 }
```

王手生成の例: template の利用 まとめ

- 手番を template 引数 Turn にして関数を定義する
- 呼び出すときに template 引数 Turn の値として、先手または後手を指定する (変数は不可)
- 王手生成先手 と 王手生成後手 を手で書いた場合と同じ効率を実現

```
template ⟨Player Turn⟩ 王手生成 (state, kingx, kingy) {
  // 歩で王手
   y = (Turn == 先手) ? kingy-2 : kingy+2;
   if ((kingx, y) にいるのが歩なら) {
      promote = (Turn == 先手) ? y<=4 : y >= 6;
      if (promote) この歩を成る手を生成
      ...      }
... 以下他の駒
```

template 上級編: partial specialization

典型例 Fibonacci:

- **一般的な定義**

```
template<int A> struct Fib
{static const int val=Fib<A-1>::val+Fib<A-2>::val; };
```

- A=1 **の場合**

```
template<> struct Fib<1> {static const int val=1;};
```

- A=0 **の場合**

```
template<> struct Fib<0> {static const int val=0;};
```

この関数はどうコンパイルされるか?

```
int fib3() { return Fib<3>::val; }
```

→Fib⟨3⟩ **は一般的な定義にマッチ**

典型例 Fibonacci

- **一般的な定義**

```
template<int A> struct Fib
{static const int val=Fib<A-1>::val+Fib<A-2>::val; };
```

- **A=1 の場合**

```
template<> struct Fib<1> {static const int val=1;};
```

- **A=0 の場合**

```
template<> struct Fib<0> {static const int val=0;};
```

この関数はどうコンパイルされるか?

```
int fib3() { return Fib<3>::val; }
Fib<3>::val = Fib<2>::val + Fib<1>::val;
```

典型例 Fibonacci

- **一般的な定義**

```
template<int A> struct Fib
{static const int val=Fib<A-1>::val+Fib<A-2>::val; };
```

- **A=1 の場合**

```
template<> struct Fib<1> {static const int val=1;};
```

- **A=0 の場合**

```
template<> struct Fib<0> {static const int val=0;};
```

この関数はどうコンパイルされるか?

```
int fib3() { return Fib<3>::val; }
  // Fib<3>::val = Fib<2>::val + Fib<1>::val;
  // Fib<2>::val = Fib<1>::val + Fib<0>::val;
```

すなわち、return (1+0)+1; **と書いた場合と同じコードが生成される。**

partial specialization の応用

ストーリー:

- 小駒の動きによる王手は動ける場所だけが違う
- 各駒で王手をかける関数をいちいち書くのは面倒
- 各駒の種類が「右上に動けるかどうか」等 10 近傍の表を用いるとコードは綺麗に。だが速度に犠牲が出る

→ 駒の種類を template 引数に!

- 王手 ⟨ 歩 ⟩(kingx, kingy); などとして利用
- PtypeTraits (osl/include/ptypeTraits.h)

templateを利用した機能の組み合わせ

状況:

- 大体同じ機能だが部分的に異なる関数が必要に
  - 相手に王手をかける手を生成
  - 相手に王手をかける手で、自玉を取られない手を生成
  - 飛車が逃げる手を生成
  - 飛車が逃げる手で、自玉を取られない手を生成

解決案と得失:

- すべての組み合わせを手で書く: 速いが面倒
- 指手を生成してから選択: シンプルだが遅い
- 関数ポインタで組み合わせ: シンプルだが遅い (*)

→ 指手を生成した後の処理をtemplate引数に: シンプルで速い

templateを利用した機能の組み合わせ

状況:

- 指手生成関数(A): 取る、逃げる、利きをつける他
- 指手の性質を見る関数(B): 自殺手か? 王手? 他
- Aで指手生成→Bでフィルタしたい

解:

- 案1: 速いがA×Bの組み合わせが必要に
  1. 王手を作りながら自殺手でないものだけ出力する関数を新たに書く
- 案2: シンプルだが多少無駄
  (駒打ちは自殺でないなどの情報を捨てる)
  1. 王手をすべて作り一時的に配列に格納
  2. for (全部の手) 自殺でない指手のみを出力

template を利用した機能の組み合わせ

template⟨typename X⟩ **指手生成** () {
　指手候補 m **を作っては、何かする** X(m)
}

呼び出すときの組み合わせにより X **を変更**:

- **指手生成⟨配列に格納⟩**()
  - **作成した指手を配列に格納する**
- **指手生成⟨自殺手を除く⟨配列に格納⟩⟩**()
  - **作成した指手から自殺手を除いた指手を配列に格納する**

ソースコード: osl/include/osl/move_action/store_.h **他**

目次

enum Player

- Player: BLACK (0), WHITE (-1) // **なぜ**-1 **か**?
- **ソースコード**: include/osl/player.h

使用例:

```
Player pl = BLACK, pl2 = alt(pl);
  // pl2 == WHITE
int v = 10 * playerToMul(pl);
  // (pl == BLACK && v = 10) | | (pl == WHITE && v = 10)
int a[2] = { 100, 200 };
int b = a[playerToIndex(pl)];
  // (pl == BLACK && b = 100) | | (pl == WHITE && b = 200)
```

template class CArray

- template ⟨typename T, size_t Capacity⟩ class CArray;
- **機能**: C **の配列**+assert **による範囲チェック**
- **ソースコード**: include/osl/misc/carray.h

使用例:

```
CArray⟨int,2⟩ a; // int a[2]; と(ほぼ)同じ
CArray⟨char,3⟩ b; // char b[2]; と(ほぼ)同じ
CArray⟨int,2⟩ c = {{5,8}}; // int c[2] = {5,8};
  // c[0] == 5, c[1] == 8
  // c[playerToIndex(BLACK)] == 5, c[playerToIndex(WHITE)] == 8
  // c[BLACK] == 5, c[WHITE] == 8
```

enum Ptype

- Ptype
  - KING, GOLD, PAWN, LANCE, KNIGHT, GOLD, SILVER, BISHOP, ROOK (EMPTY, EDGE)
  - **成ったものは** P **を先頭に** PPAWN, ... PROOK (c.f., Bonanza **は** Dragon)
  - **実装**: 8 **を引くと成る、足すと戻る**
- **ソースコード**: include/osl/ptype.h

使用例:

```
Ptype rook = ROOK, prook = promote(rook);
  // prook == PROOK
  // unpromote(PROOK) == ROOK
  // unpromote(PAWN) == PAWN
  // unpromote(GOLD) == GOLD
```

enum PtypeO

- PtypeO
  - Ptype + Player (O は Owner)
  - 実装: 正が BLACK, 負が WHITE
- ソースコード: include/osl/ptype.h

使用例:

```
PtypeO ptypeo = newPtypeO(BLACK, PROOK);
  // getPtype(ptypeo) == PROOK
  // getOwner(ptypeo) == BLACK
  // captured(ptypeo) == newPtypeO(WHITE, ROOK)
  // eval.value(newPtypeO(BLACK, PAWN))
  //  == -eval.value(newPtypeO(WHITE, PAWN))
```

class Position, Offset

- Position: file + rank (**英語微妙** Square?)
  - **内部表現**: x*16+y+1
- Offset: Position **の差**
- include/osl/{position,offset}.h, lib/offset.cc

使用例:

```
Position p27(2,7), p26(2,6);
  // p26.x() == 2, p26.y() == 6;
Offset o(0, -1), o2 = p26 - p27;
  // o == o2;
  // p27+o == p26, (p26+o) == Position(2,5);
  // p27.canPromote(BLACK) == false
  // p27.canPromote(WHITE) == true
  // p27.canPromote<BLACK>() == false
```

enum Direction

- Direction
  - 10 **近傍**: UL, U, UR, L, R, DL, D, DR, UUL, UUR,
  - **長い方向**: LONG_UL, LONG_U, ..., LONG_DR
- **ソースコード**:
  include/osl/direction,directionTraits.h

使用例:

```
// shortToLong(U) == LONG_U
// longToShort(LONG_U) == U
```

以下、普通に使うだけなら解読不要:

```
// DirectionPlayerTraits<BLACK,U>::offset()
// == -DirectionPlayerTraits<WHITE,U>::offset()
// DirectionPlayerTraits<BLACK,U>::offset() == Offset(0,1)
```

class Piece

- Piece: **駒** (Position+PtypeO **もしくは 空白 もしくは 盤外**)
- **ソースコード**: include/osl/piece.h,lib/piece.cc

使用例:

```
Piece p = 初期局面.getPieceAt(Position(7,7));
  // p.position() == Position(7,7)
  // p.ptypeO() == newPtypeO(BLACK, PAWN)
  // p.owner() == BLACK
  // p.ptype() == ROOK
```

class Move

- Move: **指手**
- **ソースコード**: include/osl/move.h,lib/move.cc

使用例:

```
Move m76(Position(7,7), Position(7,6), PAWN,
PTYPE_EMPTY, false, BLACK);
  // m76.to() == Position(7,6)
  // m76.to() == Position(7,7)
  // m76.ptype() == PAWN
  // m76.capturePtype() == PTYPE_EMPTY
  // m76.isPromote() == false
  // m76.isDrop() == false
  // m76.player() == BLACK
```

C++経験者向けクイズ

Q. OSL には PtypeO などの enum と Piece のように class がある。PtypeO も class にすると、getOwner などを大域関数からメンバ関数にすることができて使い勝手が良い。にもかかわらず、class にされていない理由はなにか?

A. 答えはそのうち

目次

将棋盤の表現

大枠: 駒番号方式 (c.f. YSS)
ソースコード:
include/osl/state/{simple,numEffect}State.{h,tcc}
データ表現:

- **駒台** CArray⟨PieceMask,2⟩ stand_mask;
- **盤面 場所 →駒**
  CArray⟨Piece,Position::SIZE⟩ board;
- **駒 駒番号 →駒** CArray⟨Piece,Piece::SIZE⟩ pieces;
- **二歩確認用** x **座標の** bitset
  CArray⟨BitXmask,2⟩ pawnMask;
- **盤面にある駒一覧** CArray⟨PieceMask,2⟩
  onBoardMask;
- **成駒一覧** PieceMask promoted;

利きの表現

ソースコード:
include/osl/effect/numSimpleEffect.{h,tcc}
利き:

- 0-39：0-39 **の利き**
- 40-47：32-39 **の長い利き**
- 48-53：**黒の利きの数** (sentinel **を合わせて** 6bit)
- 54-59：**白の利きの数** (sentinel **を合わせて** 6bit)

補足:

- **長い利きの有無は逃げる際に便利**
- **利き数→**popcount **命令のない** cpu **で有利**
- bit **を立てつつ利きの数を** 1 **増やす操作は同時に実現可能**

指手生成の実際

逃げる手: include/osl/move_generator/escape_.{h,tcc}
読んでみたい関数があれば簡単に解説します。

Q&A (後日追記)

Q. template を使わなくても inline 展開されればコンパイラが最適化してくれるのではないか?
A. GPS 将棋の場合には、inline 展開されない大きな関数でも先手と後手を分ける効果があった

Q. 共同開発はどんな分担か
A. メインで触る部分はそれぞれ傾向があるが、明確な範囲分けはない。強くした人はどこを触っても良い。

Q. gps_l の評価関数はどこか?
A. osl/include/osl/eval/experimental/testEval.h と osl/lib/eval/experimental/testEval.cc にある。
osl-for-csa ではなく OSL をダウンロードする。(スライド 6 ページ「最新ソースコード」の辺りを参照)

Q&A (後日追記)

Q. gps_l を動かしているハードウェアは何か?
A. gps_normal も gps_l は似た構成の opteron。いずれも逐次版。

Q. 今までに効果のあった特徴の例は?
A. 玉との相対位置を序盤終盤で分けるまでは鳴かず飛ばず →駒割と位置評価のバランスを取るには進行度が重要か? 最近では飛車や角が睨んでいる駒の種類と、その駒の場所と玉との関係など。

Q. 特徴は差分計算できるもののみを使っているか?
A. 2 駒の関係等差分計算するものと、玉の周り 5x5 の状態のように値が変化したら再計算するもの。

Q. 特徴を加えるたびに順調に強くなるものか
A. 自己対戦で勝ち越したもののみを採用している。が、floodgate のレートが下がると色々心配になる。

Q&A (後日追記)

Q. (J' のような) 学習の効果の指標は何を使っているか?
A. GPW2008 で使った

$$\text{不一致度} = \frac{1}{\text{局面数}} \sum_{\text{合法手 i}} T\left(\xi(\text{i}) - \xi(\text{棋譜の手})\right)$$

(ξ は最善応手手順後の局面の評価値，$T(x)$ はシグモイド関数 $T(x) = 1/(1 + \exp(-3x/128))$ で 128 は歩の点数) を使っている。(保木さんは局面の代わりに合法手の数で割った値を使っているとのこと。)

Q. どこかで見かけた J'=2.9 は本当か?
A. 棋譜次第だが初見の棋譜でも少し動かすと 3 を切るという意味では誤りではない。k-fold cross validation 等の数値ではないので、目安として扱ってほしい。

Q&A (後日追記)

Q. いわゆる「ペナルティ」は使っていないのか
A. GPW08 の時点では l_1 正則化の効果は見出せなかった。念のため出現頻度があまりに低い特徴は無視している。

Q. 1 手+静止探索を 0 手にすると問題が起こるか?
A. 逃げる手を読まないと評価が安定しないと予想。

Q. 実現確率はどのように決めているか
A. 指手の優先度を評価関数に求めた上で、どの順位の手が何%で指されるか等の統計を取って変換。root で確率が低くて枝刈される指手は少ない。

Q. mpn はどのくらいか
A. とある終盤の局面では beta cut が起こった節点で 1.157。beta cut しなかった節点で alpha 値更新が起きた場合、何手目でおきたかの平均が 2.578 。