関数型プログラミング言語と証明支援器を使った金融情報システムの開発について

今井 宜洋

有限会社 IT プランニング

平成 24 年 11 月 12 日

流れ

- 概要
- 背景、目的
- 関数型プログラミング言語 OCaml の概要
- 定理証明器 (証明支援器)Coq の導入
- 良かった点、悪かった点
- 今後の展望と課題

概要

IT プランニングはいくつかの金融情報取引のためのチャートのフロントエンド及びバックエンドを開発してきた。

概要

デモ

背景 (目的)

- FIX(金融情報交換プロトコル) などで配信されるデータを取得
- 毎秒 (またはそれ以上の頻度で) 変化するデータを取りこぼすことなく記録
- エンドユーザーを対象とした多数のクライアントにリアルタイムにデータを配信
- サマータイムへ対応する処理

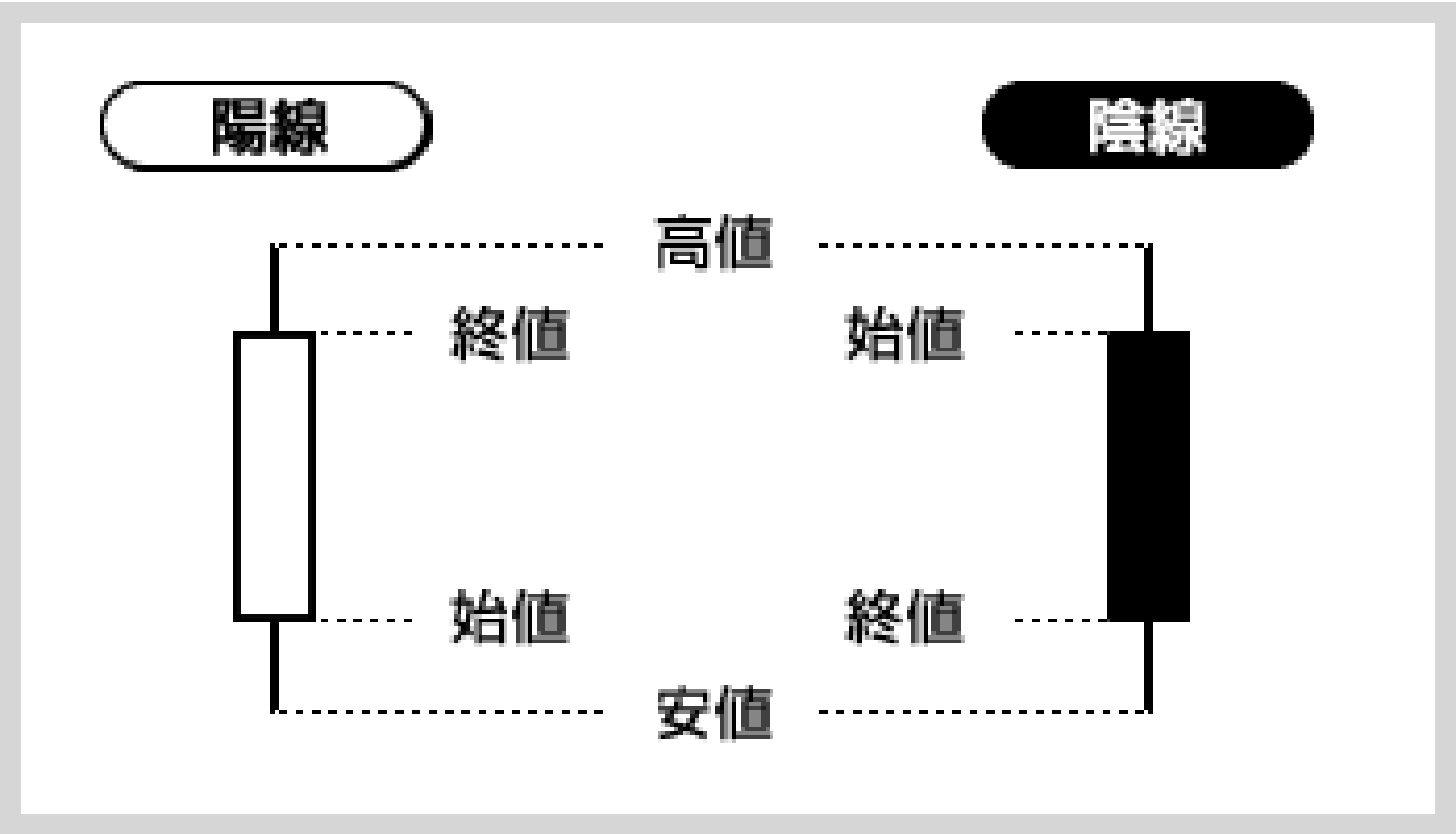
関数型プログラミング言語 OCaml の特徴

- null がない
- 強力なコンパイル時 型検査で実行時のエラーを防ぐ
- 代数データ型、高階関数で抽象的で汎用的なプログラミングができる

前提知識

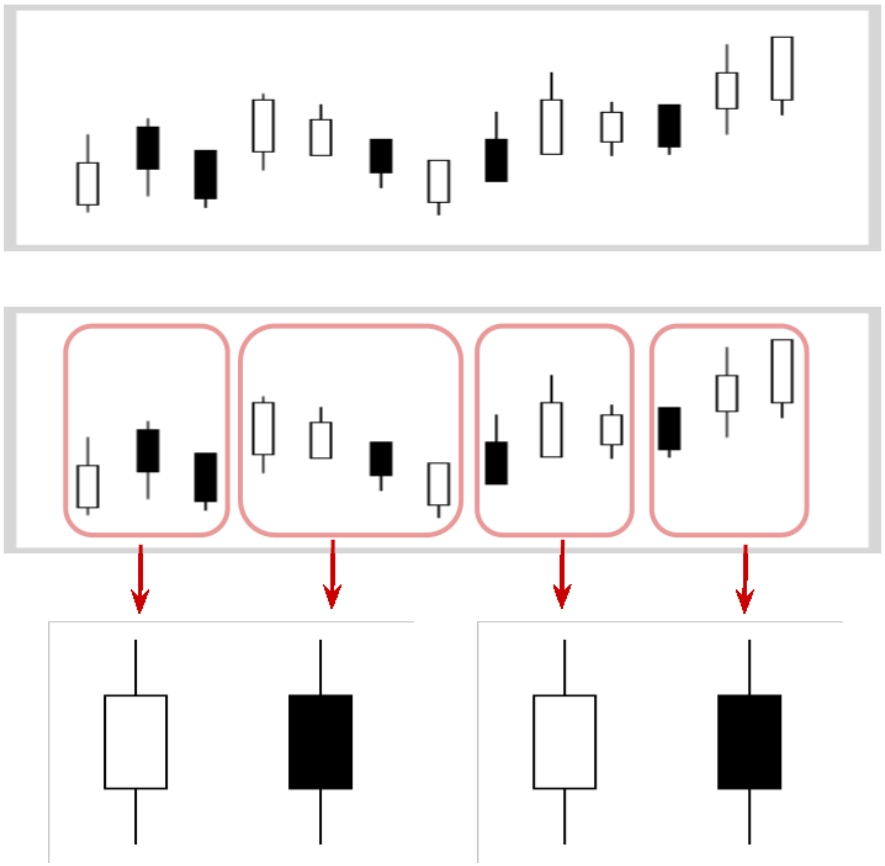
ローソク足とは

始値、高値、安値、終値の４つの値から構成される図形



ローソク足の生成フロー

小さい単位のローソク列 (データ列) からより大きい単位のローソク足を計算する



ローソク足の生成フロー

このアルゴリズムをプログラムで実際に書くと、
どのようになるか？

Java の場合

```
1  List<Candle> mergeCandles(List<Candle> candles) {
2      ArrayList<Candle> res = new ArrayList<>();
3      Date prevDate = candles.get(0).getDate();
4      Candle bigCandle = candles.get(0);
5      for (Candle c : candles) {
6          if (isSameGroup(prevDate, c.getDate())) {
7              bigCandle.merge(c);
8          } else {
9              res.add(bigCandle);
10             bigCandle = c;
11             prevDate = c.getDate();
12         }
13     }
14     res.add(bigCandle);
15     return res;
16 }
```

OCaml の場合

```
1  let merge_candles candles =
2    candles
3    |> L.groupBy is_same_group
4    |> L.map (L.reduce merge_candle)
```

OCaml の場合

```
1  let merge_candles candles =
2    candles
3    |> L.groupBy is_same_group
4    |> L.map (L.reduce merge_candle)
```

「グループ化」して、「まとめる」という流れがわかりやすい

仕様変更に対する修正

ここで、お客さんから追加修正のお願い。

仕様変更に対する修正

ここで、お客さんから追加修正のお願い。

- 終値 (getClose) が 0 より大きいもの以外は除去する
- その他の処理は同一

Java: どこを修正する?

```
1  List<Candle> mergeCandles(List<Candle> candles) {
2      ArrayList<Candle> res = new ArrayList<>();
3      Date prevDate = candles.get(0).getDate();
4      Candle bigCandle = candles.get(0);
5      for (Candle c : candles) {
6          if (isSameGroup(prevDate, c.getDate())) {
7              bigCandle.merge(c);
8          } else {
9              res.add(bigCandle);
10             bigCandle = c;
11             prevDate = c.getDate();
12         }
13     }
14     res.add(bigCandle);
15     return res;
16 }
```

Java: どこを修正する?

```
1  List<Candle> mergeCandles(List<Candle> candles) {
2      ArrayList<Candle> res = new ArrayList<>();
3      Date prevDate = candles.get(0).getDate();
4      Candle bigCandle = candles.get(0);
5      for (Candle c : candles) {
6          if (isSameGroup(prevDate, c.getDate())) {
7              bigCandle.merge(c);
8          } else {
9              if (bigCandle.getClose() > 0) {
10                 res.add(bigCandle);
11             }
12             bigCandle = c;
13             prevDate = c.getDate();
14         }
15     }
16     res.add(bigCandle);
17     return res;
18 }
```

Java: どこを修正する?

```
1  List<Candle> mergeCandles(List<Candle> candles) {
2      ArrayList<Candle> res = new ArrayList<>();
3      Date prevDate = candles.get(0).getDate();
4      Candle bigCandle = candles.get(0);
5      for (Candle c : candles) {
6          if (isSameGroup(prevDate, c.getDate())) {
7              bigCandle.merge(c);
8          } else {
9              if (bigCandle.getClose() > 0) {
10                 res.add(bigCandle);
11             }
12             bigCandle = c;
13             prevDate = c.getDate();
14         }
15     }
16     if (bigCandle.getClose() > 0) {
17         res.add(bigCandle);
18     }
19     return res;
```

OCaml の場合

```
1  let merge_candles candles =
2    candles
3    |> L.groupBy is_same_group
4    |> L.map (L.reduce merge_candle)
```

OCaml の場合

```
1  let merge_candles candles =
2    candles
3    |> L.groupBy is_same_group
4    |> L.filter (fun cs -> (L.last cs).close > 0)
5    |> L.map (L.reduce merge_candle)
```

定理証明器 (証明支援器) との連携

- 証明支援器 Coq と OCaml の連携[1]

```
1 (* NotStrideClose t1 t2 ならば、その間は常にオープンである。 *)
2 Theorem soundness : forall t1 t2 : unixtime,
3   t1 <= t2 -> NotStrideClose t1 t2 ->
4   (forall t, t1 <= t /\ t <= t2 -> IsOpen t).
```

```
1 (* StrideClose t1 t2 ならば、それらの間にクローズな時がある。 *)
2 Theorem completeness : forall t1 t2 : unixtime,
3   t1 <= t2 -> StrideClose t1 t2 ->
4   (exists t, t1 <= t /\ t <= t2 /\ ~ IsOpen t).
```

定式化及び証明コードは全部で 220 行

[1] Coq Extraction: http://coq.inria.fr/refman/Reference-Manual027.html

良かった点

- 運用の安定性 (実行時の間違いが起きにくい)
- 単体テストが適用しやすい (多くの処理は細かい関数の組み合わせで表現されるため)
- 複数人開発、他人がメンテナンスしやすい
  - 「コンパイルが通ったら多くのミスは防げる」
  - 「一部を変更したときに影響する部分をコンパイラが示してくれる」
  - 「他人のコードでも安心して変更できる」

悪かった点

- ライブラリが少ない
- 遅延評価による計算量の見積りが難しい
- (できる人が少ない!?)

悪かった点

- ライブラリが少ない
- 遅延評価による計算量の見積りが難しい
- (できる人が少ない!?)
  - 近年増えている
  - 弊社ではアルバイト含め全員がある程度はできる

今後の展望と課題

- 今回の利点は多くの静的型付け関数型プログラミングで共通
- 多くの言語が関数型プログラミングの機能を導入しつつある
- 単体テストのやり方
- 証明支援系との連携