プロパティベーステスト in JuliaLang

東京大学物性研究所
本山裕一

JuliaLang Japan 2025 @ 東京科学大
2025-12-13

自己紹介

- 専門は計算物理・統計物理・物性物理
  - これらに関するオープンソースソフトウェア開発をしている
- 2013年ぐらい(v0.2時代)からJuliaを書いていた
  - 最近は物理界隈でJuliaが流行っているのを感じている
  - 一方で自分は最近Juliaを書く機会が減っている……
- 今日は **プロパティベーステスト** について話します
  - 最近自分の中で興味が湧いてきたため
  - 研究発表ではなく、話題提供ぐらいの軽い内容です
    - コード以外は一般論なので別言語でも役立つ

ソフトウェアテストとは

- ソフトウェアや関数が要求・仕様を満たしているか確認するためにテストを行う
  - 計算速度やメモリ使用量、セキュリティなどの品質も確認する
- ソフトウェアの改善や拡張を行う場合には、既存の機能を壊さないことが重要
  - 以前と同じ入力に対して同じ出力を返すことを確認する
  - 以前成り立った性質がそのまま成り立っていることを確認する
- 研究でもテストは役立つ
  - 未知の領域に挑む前に、既存の成果を再現するのが大事
- 今日はブラックボックステストについて話します

ブラックボックステスト

- ソフトウェア全体や、関数などの部品はそれぞれ、入力 x を受け取り、出力 y を返す関数 $y = f(x)$ として扱える
  - 副作用なども含めて出力とみなす
  - 今回はテスト対象のことをまとめて関数 f と呼ぶ
- 関数内部の実装の詳細には立ち入らず、入出力の関係をテストするのがブラックボックステスト
  - 実際に動かしながらテストを行っていく

事例ベーステスト (EBT)

- 具体的な入力 x と期待される出力 y の組（テストケース）を用意
  - $y' = f(x)$ が期待される出力 y になっているかを確認
- Juliaだと標準ライブラリ `Test` で簡単にかける

```
using Test
@testset begin
  @test sort([]) == []
  @test sort([3, 1, 2]) == [1, 2, 3]
end
```

EBTの難点

- 用意したテストケースしかテストできない
- 網羅的にテストケースを作るのは技術が必要で、手間がかかる
  - 同値クラス・境界値・デシジョンテーブルなどのテスト設計技法
  - 分業できる専門のテストチームがいればいいが、自分でやるとなかなか大変
    - 妥協されることが非常に多い
- テストケースを自動生成すると手間が軽減されるのでは？
  - → **プロパティベーステスト(PBT)**

`Supposition.jl`

- PBTを実現するパッケージ
  - https://github.com/Seelengrab/Supposition.jl
- Python のPBTパッケージ `Hypothesis` に影響を受けている
- 同じ作者が作った `PropCheck.jl` の後継パッケージ
  - こちらはHaskellの `Hedgehog` に影響を受けている

プロパティベーステスト (PBT)

- 入力 x を自動的に生成してテストする
  - x に対応する、期待される出力 y は **わかりようがない**
  - かわりに、y が満たすべき **性質（プロパティ）** を考える
    - 関数の **事後条件** とも言える
    - 例：ソート関数の場合、出力は **ソート済み**
- 大量に自動生成された入力に対して計算された $y' = f(x)$ がプロパティを満たすかをテスト
  - 全部通れば、f がそのプロパティを満たしているとみなせる

入力の制御

- 関数は事前条件をもつ
  - 入力 x が満たすべき条件
    - 例：ソート関数の場合、入力は配列である
  - 事前条件を満たすのは呼び出し側の責任
- `Supposition.jl` では入力の生成器 Generator を柔軟に作成できる
  - 例：すべての要素が-100以上100以下の整数であり、長さが3から10の配列

```
generator = Data.Vectors(Data.Integers(-100, 100), min_size=3, max_size=10)
```

例：ソート関数のプロパティ

- ソート関数の事後条件
  - 出力 y はソート済み配列である
    - つまり、 `issorted(y)` が `true` となればOK
- `Supposition.jl` では、入力 x に対して真偽値を返す関数をプロパティとして扱う

```
function prop_sorted(x)
  y = sort(x)
  return issorted(y)
end
```

- `Supposition.@check` マクロでプロパティをテストできる
  - 引数には生成器を渡す

```
@testset begin
  @check prop_sorted(generator)
end
```

```
Test Summary: | Pass  Total  Time
test set      |    1      1  0.6s
```

- なお、 `@check` マクロ中で直接プロパティ関数を定義できる
  - その時には生成機をデフォルト引数のような形で渡す

```
@check prop_sorted(x=generator) = issorted(sort(x))
```

- ソート済みかどうかだけでは不十分
  - 空配列を返しても「単調性」は満たせてしまう！

```
@testset begin
  @check function prop_sorted(x=generator)
    y = []
    return issorted(y)
  end
end
```

- 様々な事後条件をプロパティとしてテストしていく必要がある
  - 前後で要素の集合が一致する

一般的なプロパティの例

- **冪等性**
  - 一度ソートしたものをもう一度ソートしても結果は同じ
    - `sort(sort(x)) == sort(x)`
- **逆演算**
  - エンコードした情報をデコードしてもとに戻せる
    - `decode(encode(x)) == x`
  - 配列の反転は自分自身が逆演算
    - `reverse(reverse(x)) == x`

- **参照実装**
  - 正しいことが確認しやすい別実装を用意する
    - `merge_sort(x) == bubble_sort(x)`
  - 既存実装の高速化や省メモリ化
    - `f_new(x) == f_old(x)`
  - 別言語・別プログラムからの移植
  - 成り立つ範囲が狭いが正しい実装
    - より強い仮定のもとで解析解が存在する場合など

プロパティを考える利点

- 一般的に、 **検証するほうが実装するより簡単**
  - 「ソートされているかを調べる関数」は簡単に書ける
  - それに比べると「(高速に）ソートを行う関数」は複雑
- 検証する関数（プロパティ関数）は賢く書けなくとも良い！
  - あくまでテストで使うだけなので、遅くても良い
  - もちろんテストが短くなるならそれに越したことはない
- 先にプロパティを書くのも有効
  - **プロパティ駆動開発**：テスト駆動開発のプロパティ版

反例の収縮(shrinking)

- 反例が見つかったら、デバッグのために反例を分析する必要がある
  - 自動生成されているので、そのままだと人間にとってよくわからない反例が表示されるのでは？という自然な懸念
- プロパティベーステストの重要機能として **反例の収縮** がある
  - 一度反例を見つけたら、入力を小さくしながらプロパティを確認し、できるだけ単純な反例を探す
  - `Supposition.jl` では自動でやってくれるので実際には気にしなくて良い

PBTの利点

- **全自動で多数の入力例を生成**
  - 多くの入力パターンを自動生成することで、漏れを減らせる
- **仕様の理解が深まる**
  - プロパティを考えることで、仕様をより深く理解できる
  - 深く理解する必要がある、とも言える

PBTの欠点

- **EBTよりも時間がかかる**：
  - 大量の入力を試すことでテストの漏れを減らす手法
    - `Supposition.jl` のデフォルトでは10000個の入力を試す
  - アプリ全体など、重い対象には不向き
    - 適宜サンプル数を減らすとよい

```
@check max_examples=10 prop_sorted(generator)
```

まとめ

- **ソフトウェアテストは重要な技術**
  - 事例ベーステストは、具体的な入力と期待される出力をテストケースとして用意する
    - 事例を一つ一つ作る必要がある
  - **プロパティベーステスト**は、自動生成された入力に対して、期待される出力が満たすべき性質をテストする
    - 事前条件や事後条件といった仕様をより深く理解できる（理解する必要がある）
- Juliaでは `Supposition.jl` でとても簡単にプロパティベーステストを行える
  - 入力の生成機を柔軟に作成できる
  - プロパティの定義も簡単
  - 既存のテストに組み込むのも簡単