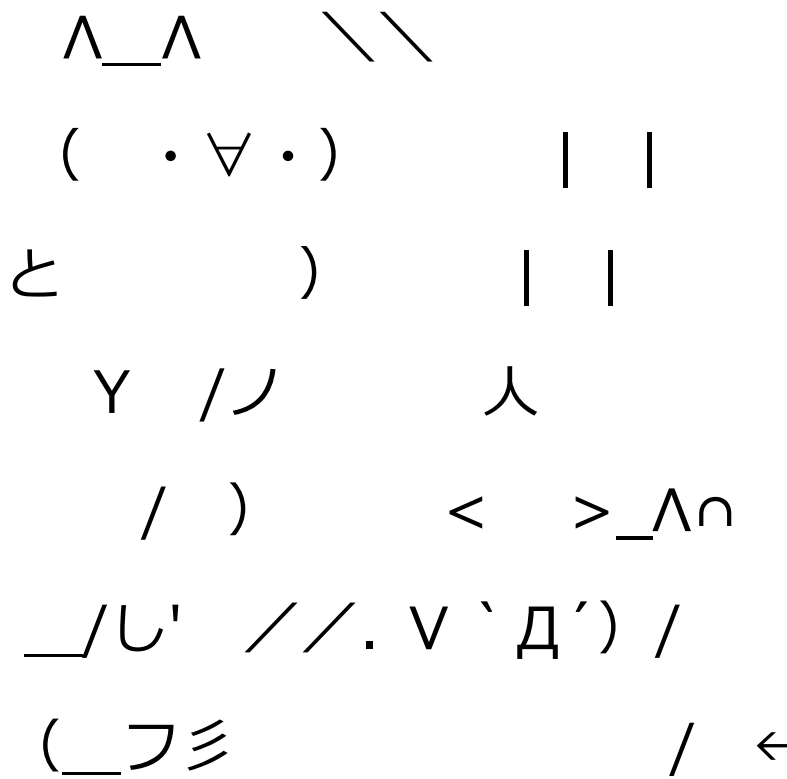
プログラマは
ぬるぽ静的解析の夢を見るか？

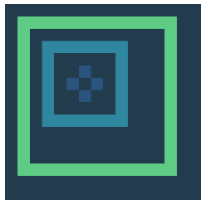
Show（@ajfAfg）

^__^
(　´∀｀) ＜ぬるぽ



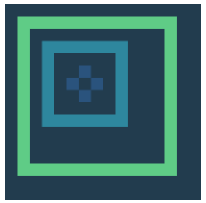
ガッ

自己紹介



- Show（真名は佐々木勝一）
- 24卒の生産性向上エンジニア
- 好きなもの
  - プログラミング言語（型、形式的意味論）
  - 音楽（ダンスミュージック、ゲーム音楽（、ジャズ））
  - お酒（特に、ビール、日本酒）
  - 体を動かすこと

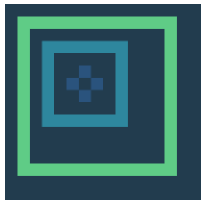
ある日のShow



NilAwayってツール
面白いな！

この辺気になってたし
勉強がてら発表するか！

ある日のShow



ぬるぽと言えば
10億ドル損失の
話題やな！

11/18に出した応募内容↓

タイトル：プログラマはぬるぽ静的解析の夢を見るか？（仮）
概要：null の発明者であるアントニー・ホーアが「Null References: The Billion Dollar Mistake」と話しているように、ぬるぽは開発者だけでなくお客様の損害まで繋がります。そのため、開発段階でぬるぽを排除したいです。本発表では、ぬるぽを検出する静的解析手法やその難しさをはじめ、Java や Go での実例を紹介します。（仮）

先行研究

https://speakerdeck.com/ytaka23/kyoto-go-56th

スライド ショーの最後です。クリックすると終了します。

めげずに、ぬるぽ静的解析の世界を
もっと俯瞰的に話します

ぬるぽ検出のモチベ

- 10億ドルの損失を生むから
- 「java.lang.NullPointerException」見たくないですよね！！

素朴なぬるぽ静的解析

- Dereferenceする箇所からプログラムを遡って
  nullが入るか調べたらいい
- 簡単😁

厳しい現実😫

- 実は、ぬるぽの静的解析は一般に決定不能*
- どの解析手法も何かを諦めてる
  - 偽陰性・偽陽性を許容
  - 解析にめっちゃ時間かかる
  - 言語のサブセットのみ解析可能

問題を解くアルゴリズムが
存在しない的な意味

* G. Ramalingam. 1994. The undecidability of aliasing.
ACM Trans. Program. Lang. Syst. 16, 5 (Sept. 1994), 1467–1471. https://doi.org/10.1145/186025.186041

スライド ショーの最後です。クリックすると終了します。

完璧でなくても
便利な手法いろいろあります

TypeScriptの型システムが型安全
じゃないけど便利なのと似てる

データフロー解析で頑張る（1/3）

- 素朴なぬるぽ解析の延長的な解析手法で、変数に入る値を伝播していく
- ツール例: NilAway (Go), NullAway (Java), FindBugs (Java)

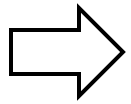
```
// {}
String foo = null;
// { foo=null }
String bar = "シュタゲいいよね";
// { foo=null, bar=not null }
foo.length(); // エラー！！！！！
```

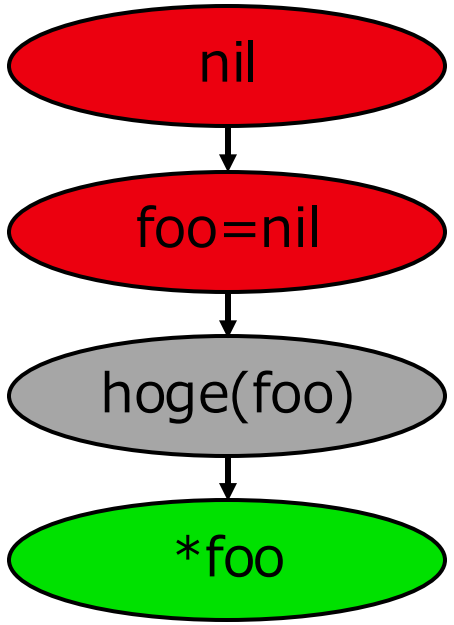
変数の中身が伝播してる

データフロー解析で頑張る（2/3）

- もっとクレバーに解析するNilAway
- 2-SATという計算問題に帰着できて、これは線形時間で解ける

```
package main

func main() {
    hoge(nil)
}
func hoge(foo *int) int {
    return *foo
}
```





赤背景: nil
緑背景: not nil
灰背景: 矛盾

データフロー解析で頑張る（3/3）

- 典型的な弱点: 配列の要素まで追跡しない

Goでは関数の初期値はnil

```go
package main

func main() {
    var arr [2]func() int
    arr[0]()
}
```

```
$ nilaway ./...
# 何もなし

$ go run .
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x77380]

goroutine 1 [running]:
main.main()
        /app/main.go:5 +0x20
exit status 2
```

分離論理で頑張る（1/2）

- 分離論理とはメモリを表現可能な論理体系
  - 魔法の杖（−∗）という演算子が出てきてメルヘン
- ホーア論理と組み合わせてプログラム検証する
  - ホーア論理の例: { x = 1 } x = x + 3 { x = 4 }

事前条件　**プログラム**　**事後条件**

分離論理で頑張る（2/2）

- ツール例: Infer (Java, C++, Objective-C, C)
- 任意の関数の事前条件と事後条件を推論（Bi-abduction）

前提: { x ↦ – } `void use_cell(int *x)` { emp }

ポインタxはある値を指す

メモリは未使用

推論対象:

```
void g(int *x, int y) {
    y=0;
    use_cell(x);
}
```

`y=0;` ← 関数に閉じた変更なので無視

`use_cell(x);` ← `use_cell`の事前・事後条件をgに伝播

⇨ 推論結果:
{ x ↦ – } `void g(…)` { emp }

完璧なぬるぽ静的解析は無理！
どうしても漏れが出る！！

救いはないのか！？

Option型という救世主

- nullかどうかが型にエンコードされているので解析しやすい
- 高速度や省メモリが求められないなら使っとくと便利

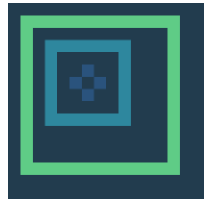
```ts
type Option<T> = { _tag: "Some", value: T } | { _tag: "None" }

const get = (foo: Option<string>, defaultValue: string): string => {
    // NG
    return foo.value;

    // OK
    if (foo._tag === "Some") return foo.value;
    else return defaultValue;
}
```

Property 'value' does not exist on type '{ _tag: "None"; }'.

Option型最高！！！！

ウチにもOption型
あるよ

最高！！



ウチにもOption型
あるよ

ただしユーザー定義

Option型1

Option型2

Option型3

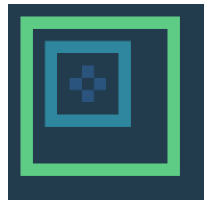
Option型4

Option型5

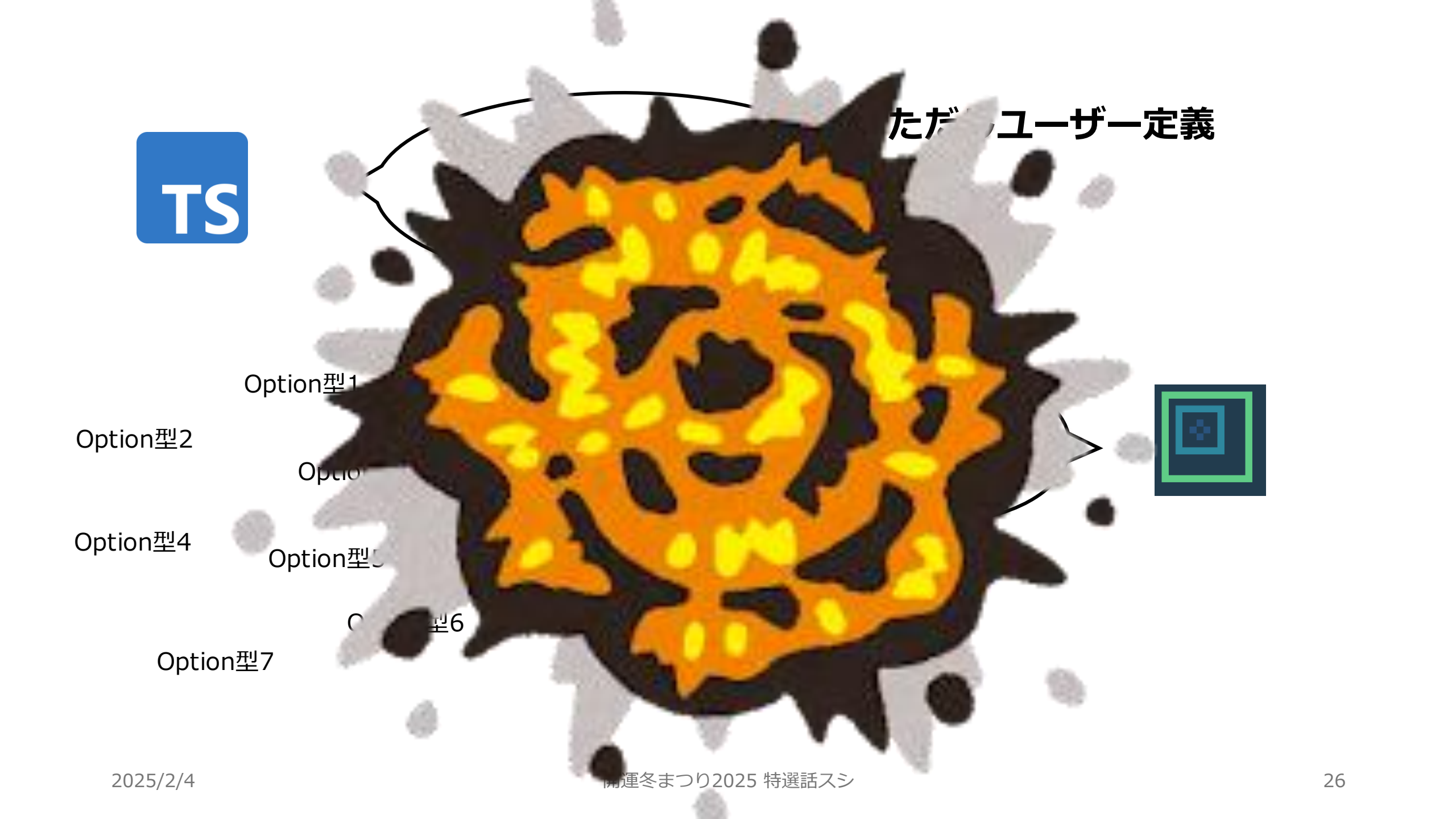
Option型6

Option型7

最高！！



各々が定義したOption型
互換性ない


ただ ユーザー定義
Option型1
Option型2
Option型4
Option型7

まとめ

- ぬるぽ静的解析は一般に決定不能
- 完璧じゃないけど便利な手法がいろいろある
  - データフロー解析
  - 分離論理
- できればOption型使いたい