RustBelt ゼミ第1回
Chapter 1, 2

K.Hirata

Chapter 1
Introduction

- ほとんどの低レベルプログラミングは C か C++ で行われている。
- Microsoft や Chrome team はセキュリティの脆弱性の約70%がメモリ安全性がの侵害によって引き起こされている、と報告している。
- 他のたくさんの言語で、ある種のメモリに対する control を放棄することで、メモリ安全性を確保することに成功しているものたちはある
    - (例) GC, データレイアウトや deallocation をプログラマがさわれないようにする
- 一方で、システムプログラミングなどの、資源の consumption (CPU 利用時間・使用メモリ量など) を最小化することが非常に大事になる場面では、C や C++ と同様なレベルで control を持たねばならない。

Rust

- Rust は上記の課題を解決する。
  - メモリマネジメントに関して、C++ と同等に強力
  - 型安全性とメモリ安全性を担保
    - Vector の要素に操作を行うときなども、言語が書き換え可能性に関して厳しく規定し、バグを防ぐ。
    - スレッドが複数あるときに、shared mamory を通した意図しない data race を起こさせない。
    - すごい！w

1.1 Making systems programming safer with unsafe code?

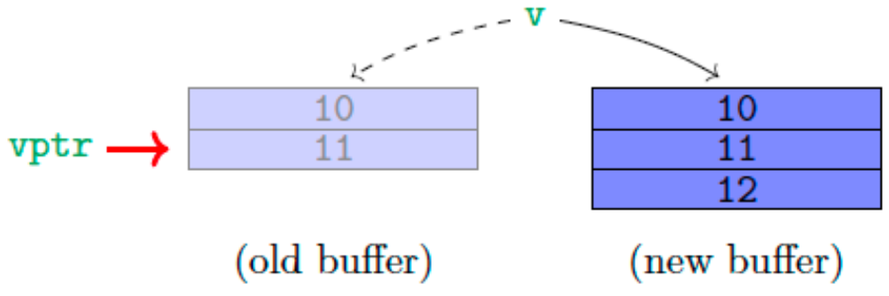
- C++ の次のコードを考える

```
1  std::vector<int> v { 10, 11 };
2  int *vptr = &v[1]; // Points *into* ‘v‘.
3  v.push_back(12);
4  std::cout << *vptr; // Bug (use-after-free)
```

- Vector v = [10, 11] を作って、11 のところへのポインタ vptr をとる。その後 v の末尾に 12 を追加して v = [10,11,12] にする。最後に、vptr の指す値を出力する。

(おそらく確率的に) Panic: 既に free されています

解説:



何が悪かったか

```
1  std::vector<int> v { 10, 11 };
2  int *vptr = &v[1]; // Points *into* ‘v‘.
3  v.push_back(12);
4  std::cout << *vptr; // Bug (use-after-free)
```

- これは Iterator invalidation と呼ばれるバグの一種。

- よくある例は、ある container (ここでは vector) の中身を loop などで繰り返し触る際に、間接的かつ accidentally に データ構造を変更してしまうことで発生する。

- この場合、(読み取り可能) ポインタ vptr の生存区間内で、(書き換え可能) ポインタ v が書き換えを行う関数 push_back へ引き渡されたことが悪い。

- Rust では、次のようなプログラムに相当するが、これはコンパイル時の静的解析でエラーが検知される。

```
1  let mut v = vec![10, 11];
2  let vptr = &mut v[1]; // Points *into* ‘v‘.
3  v.push(12); // May reallocate the backing store of v.
4  println!("{}", *vptr); // Compiler error
```

“cannot borrow v as mutable more than once at a time”

ライフタイム、region、所有権、etc.

より低レベルな表現

- Rust は、上の例のように ”表面上” 安全な型システムを持っている。すなわち、Vector を使う上で**ライブラリを呼び出して使う分には**、C++ などでは検知できなかったポインタのアクセス権限などに関して、多くのバグ検出を行える。

- 一方で、そのライブラリがどう書かれているかというと、ライブラリの内部では **unsafe** キーワードを用いて、そのような型による保証のない領域に書かれている。

    - 低次すぎてそもそも保障ができない。

        - ゴリゴリの高速化のためにコンパイラで保障された安全性を捨てる。

- Vector crate 自体の安全性はわからないけど、「crate Vec が正しいならこのプログラムはメモリ安全である。」という保障。

- Vec 以外にも、Mutex (mutual exclusion をやる crate) なども、同様の問題がある。実際標準ライブラリからバグが見つかったり、ライブラリの API が提供する型では保障が足りなかったりする問題は 0 ではない。unsafe が関わる部分では、型システムに関する保障では足りない部分が多々ある。

1.2 Understanding Rust: RustBelt

- For this dissertation, we have developed RustBelt, the first formal (and machine-checked) model of Rust which can account for Rust's extensible approach to safe systems programming and verify its soundness.

- RustBelt は machine-checked とは言いつつ、人力で Coq まで落とし込むらしい。(Coq が正しければ、元のソースも正しい)

https://twitter.com/blackenedgold/status/1209110852864897024?s=20&t=R0Yt7mYlSrjkSy5ALbMn1Q

- Rust はいくつかのコンパイラ内中間表現を持っていて、ソース → AST → HIR → THIR → MIR → LLVMIR と変換するが、このうちの MIR が言語のモデルとして扱いやすい。実際これの core をモデル化して、λRust を定義する。(MIR でも結構 Rust の特徴は残っているので、問題ない)

RustBelt の概要

With our semantic model in hand, the safety proof of λ_{Rust} divides into three parts:

1. Verify that the typing rules of λ_{Rust} are sound when interpreted semantically. For each typing rule we show a lemma establishing that the semantic interpretations of the premises imply the semantic interpretation of the conclusion. This is called *the fundamental theorem of logical relations.*
2. Verify that, if a closed program is *semantically* well-typed according to the model, it is safe to execute—there will be no safety violations. This is called *adequacy.*
3. For any library that employs **unsafe** code internally, verify that its implementation satisfies the "safety contract" of its public interface. This safety contract is determined by the semantic interpretations of the types of the library's public API. It establishes that all **unsafe** code has been properly "encapsulated" behind that API. In essence, the semantic interpretation of the interface yields a *library-specific verification condition.*

1 の証明は Coq で行われている。Arc, Rc, Cell

(§13.1), RefCell, Mutex (§13.2), RwLock, mem::swap, thread::spawn, rayon::join, and take_mut などを含むライブラリの verification を行なった。

Choosing the right logic for modeling Rust

- λRust の semantic model として、所有権もちゃんと考慮されないといけない。そこで sepqration logic を利用した。

- separation logic には **points-to** (\mapsto 記号) と、**separating conjunction** (∗) があって、これを使って Vec などの所有権の概念を自然に落とし込める。

- ただし普通の separation logic ではかなり制限された所有権しか表現できない。そこで、**Iris** という強い logic/frame work を使う。Iris の特徴は、**step-indexing** というものがサポートされていることで、これが難しくてややこしいらしい。あまり綺麗じゃなくて好きではない、と指圧マットさんはおっしゃっていた。

- Iris 自体が Coq の上に正当化されている。

Modeling lifetimes and borrowing

- Iris は Iris 独自の “所有権” の概念を持っているが、λRust の完全なモデル化には、「あるライフタイムでの借用 (borrow something for a certain lifetime)」を表現する必要がある。

- Using all key features of iris, including **impredicative invariants** and **higher-order ghost state**, we construct the novel **lifetime logic** and verify its correctness in Iris.

1.3 Evolving Rust: Stacked Borrows

```rust
fn example1(x: &mut i32, y: &mut i32) -> i32 {
  *x = 42;
  *y = 13;
  return *x; // Has to read 42, because x and y cannot alias!
}
```

- この関数はいつも 42 を返すので、最適化して return 42; に変えてしまいたい。
- 実はこれは嘘で、次のような呼び出し方ができる。

```rust
fn main() {
  let mut local = 5;
  let raw_pointer = &mut local as *mut i32;
  let result = unsafe {
    example1(&mut *raw_pointer, &mut *raw_pointer)
  };
  println!("{}", result); // Prints "13".
}
```

- 実際 RustBelt 執筆当時の rustc 1.35.0 では、13 を返す。

- In the third main part of this dissertation, we describe **Stacked Borrows**, an operational semantics for pointer aliasing in Rust. Stacked Borrows says that our example program has **undefined behavior**, which means the compiler is allowed to compile it in any way it chooses.

  - To ensure that Stacked Borrows does not introduce "too much" undefined behavior, we have equipped Miri,[43] an existing interpreter for Rust programs, with an implementation of Stacked Borrows. We have run the OS-independent part of the Rust standard library test suite[44] in this interpreter. The majority of tests passed without any adjustments. We also uncovered a few violations of the model, almost all of which have been accepted by the Rust maintainers as bugs and have since been fixed in the standard library. For further discussion, see §18.
  - To ensure that Stacked Borrows has "enough" undefined behavior, we give proof sketches demonstrating that a few representative compiler transformations, such as the one in `example1`, are legal under this semantics. These proof sketches are all backed by fully mechanized proofs in Coq,[45] based on a formalization of Stacked Borrows and a framework for open simulations.[46]