Linear Types とプログラミング言語 Rust について

長谷川 央

2024-05-12

プログラミング言語 Rust

Rust は最近流行り（？）のプログラミング言語

C/C++の代替としての使用が期待されており、Linux カーネルでも一部、導入が始まっている

Rust の強み

- コンパイル時にメモリ安全性が保証される（バッファオーバーフローなどが起きない）
- ガベージコレクションを使用しないため、高速である
- 統合ビルドツール Cargo が使いやすい
- 非同期プログラミングなど、モダンな機能がサポートされている

プログラミング言語 Rust の特徴

- 所有権
  一度に一つの変数だけが値を所有できる
- ムーブ
  値は別の所有者へ所有権を移すことができる
- 借用
  値を借用することで、所有権を移さずに、一時的に値にアクセス可能にする
  値を変更可能な借用は一度に一つだけ行える
  値を変更しない借用は複数個、同時に存在していても良い

Rust のコードの例：所有権

```
fn main() {
  let s = String::from("hello"); // sが"hello"の所有権を持つ
  takes_ownership(s); // sの値が関数にムーブされる
  // 下の行はコンパイルエラー。sの所有権はもう存在しない
  // println!("{}", s);
}

fn takes_ownership(some_string: String) {
  println!("{}", some_string);
}

```

Rust のコードの例：ムーブ

```
1   fn main() {
2     let s1 = String::from("hello");
3     let s2 = s1; // s1からs2へ所有権がムーブされる
4     // println!("{}", s1); // <- この行はコンパイルエラー。s1はもう有効ではない
5     println!("{}", s2); // s2は有効
6   }
7
```

Rust のコードの例：借用

```rust
fn main() {
  let s1 = String::from("hello");
  let len = calculate_length(&s1); // s1の参照を渡す
  println!("The length of '{}' is {}.", s1, len); // s1は依然として使用可能
}

fn calculate_length(s: &String) -> usize { // sはStringの参照
  s.len()
}

```

疑問（今回の主題）

Rust の**所有権**は何の理論がベースになっているのか？

Rust は Affine Type Systems に影響を受けている

この論文の Keywords に Affine type systems と書かれていた。

N. D. Matsakis and F. S. Klock, “The rust language,” in Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, in HILT ’ 14. New York, NY, USA: Association for Computing Machinery, Oct. 2014, pp. 103 – 104.

本文中に Affine type systems との関係性に関して、言及はなかったものの、
影響を受けていることは間違いない。

Affine type systems とは？

Affine type systems は、
全ての変数が高々一度しか使用されないことを保証する型システムである

別の言い方をすれば、Linear type systems に弱化規則を導入した型システムである

弱化規則の適用の例

$$\frac{x : T \vdash x : T}{x : T, y : U \vdash x : T}$$

参考：https://en.wikipedia.org/wiki/Substructural_type_system

Linear type systems とは？

Linear type systems は、
全ての変数が必ず 1 度のみ使用されることを保証する型システムである

論文では、この型システムの変数について次のように説明されている。

> 「世界」をモデリングするとしたときに、以下の 2 つの操作が許されているのは、直感に沿わない
>
> - 複製：`copy a = (a, a)`
> - 破棄：`kill a = ()`
>
> *Neither of these correspond to our intuitive understanding of “the world”: there is one world, which (although it may change) is neither duplicated nor discarded.*
>
> *Values belonging to a linear type must be used exactly once: like a world, they can be neither duplicated nor discarded.*

参考：P. Wadler, “Linear Types can Change the World!,” Program Concept Method, p. 561 –, 1990.

Conventional Types と Linear Types の比較

従来の型システムでは、変数は何度使用しても良い

$$x : Arr \vdash () : Unit$$
$$x : Arr \vdash (x, x) : Arr \times Arr$$

Linear Types では、変数は一度のみ使用可能であるため、上の式は導出できない
次の式のみが導出可能である

$$x : Arr \vdash x : Arr$$

参考：Conventional typing rules

$$\textsc{var}\ \frac{}{A,\ x : T \vdash x : T}$$

$$\to\mathcal{I}\ \frac{A,\ x : U \vdash v : V}{A \vdash (\lambda x : U.\ v) : U \to V}\ x \notin A$$

$$\to\mathcal{E}\ \frac{A \vdash t : U \to V \qquad A \vdash u : U}{A \vdash (t\ u) : V}$$

$$\boxed{K = \cdots \mid C\ T_1\ \ldots\ T_k \mid \cdots}$$

$$K\mathcal{I}\ \frac{\begin{array}{c} A \vdash t_1 : T_1 \\ \cdots \\ A \vdash t_k : T_k \end{array}}{A \vdash (C\ t_1\ \ldots\ t_k) : K}$$

$$\boxed{K = C_1\ T_{11}\ \ldots\ T_{1k_1} \mid \cdots \mid C_n\ T_{n1}\ \ldots\ T_{nk_n}}$$

$$K\mathcal{E}\ \frac{\begin{array}{c} A \vdash u : K \\ A,\ x_{11} : T_{11}, \ldots, x_{1k_1} : T_{1k_1} \vdash v_1 : V \\ \cdots \\ A,\ x_{n1} : T_{n1}, \ldots, x_{nk_n} : T_{nk_n} \vdash v_n : V \end{array}}{A \vdash (\textbf{case}\ u\ \textbf{of}\ C_1\ x_{11}\ \ldots\ x_{1k_1} \to v_1 \mid \cdots \mid C_n\ x_{n1}\ \ldots\ x_{nk_n} \to v_n) : V}\ x_{ij} \notin A$$

$$\textsc{fix}\ \frac{A \vdash t : T \to T}{A \vdash (\mathit{fix}\ t) : T}$$

参考：Linear typing rules

$$\textsc{Var}\ \frac{}{x : T \vdash x : T}$$

$$\multimap\mathcal{I}\ \frac{A,\, x : U \vdash v : V}{A \vdash (\mathrm{i}\lambda x : U.\ v) : U \multimap V}\ x \notin A$$

$$\multimap\mathcal{E}\ \frac{A \vdash t : U \multimap V \qquad B \vdash u : U}{A, B \vdash (\mathrm{i}t\ u) : V}$$

$$\boxed{\mathrm{i}K = \cdots \mid \mathrm{i}C\ T_1\ \ldots\ T_k \mid \cdots}$$

$$\mathrm{i}K\mathcal{I}\ \frac{\begin{array}{c} A_1 \vdash t_1 : T_1 \\ \cdots \\ A_k \vdash t_k : T_k \end{array}}{A_1, \ldots, A_k \vdash (\mathrm{i}C\ t_1\ \ldots\ t_k) : K}$$

$$\boxed{\mathrm{i}K = \mathrm{i}C_1\ T_{11}\ \ldots\ T_{1k_1} \mid \cdots \mid \mathrm{i}C_n\ T_{n1}\ \ldots\ T_{nk_n}}$$

$$\mathrm{i}K\mathcal{E}\ \frac{\begin{array}{c} A \vdash u : K \\ B,\, x_{11} : T_{11}, \ldots, x_{1k_1} : T_{1k_1} \vdash v_1 : V \\ \cdots \\ B,\, x_{n1} : T_{n1}, \ldots, x_{nk_n} : T_{nk_n} \vdash v_n : V \end{array}}{A, B \vdash (\textbf{case}\ u\ \textbf{of}\ \mathrm{i}C_1\ x_{11}\ \ldots\ x_{1k_1} \to v_1 \mid \cdots \mid \mathrm{i}C_n\ x_{n1}\ \ldots\ x_{nk_n} \to v_n) : V}\ x_{ij} \notin B$$

Linear typing systems を使用するメリット・デメリット

Linear typing systems を使用するメリットは、Rust を使用するメリットに近い

Linear typing systems を使用するメリット

- 各変数は一度だけ使用される。
  つまり、一度、使用した変数を自動的にフリーすれば良いので、
  reference counter やガベージコレクションが必要ない
- 同様の理由で、dangling pointer などのメモリに関する問題が発生しない
- コピーをしないため、パフォーマンスが高いまま保たれる

Linear typing systems を使用するデメリット

- 値を共有できない

Non Linear Typing Systems の導入

Non Linear な型を導入することで、前述の欠点はなくすことができる
Non Linear Types は、複製も破棄もできる型である
ただし、nonlinear なデータ構造は、linear なデータを含んではいけないという制約はある

この Non Linear Types が Rust の借用に近い概念であると考えている

参考：Non Linear typing rules

$$\textsc{kill}\ \frac{A \vdash u : U}{A, x : T \vdash u : U}\ \text{nonlinear } T$$

$$\textsc{copy}\ \frac{A, x : T, x : T \vdash u : U}{A, x : T \vdash u : U}\ \text{nonlinear } T$$

$$\to\mathcal{I}\ \frac{A, x : U \vdash v : V}{A \vdash (\lambda x : U.\ v) : U \to V}\ x \notin A, \text{nonlinear } A$$

$$\to\mathcal{E}\ \frac{A \vdash t : U \to V \qquad B \vdash u : U}{A, B \vdash (t\ u) : V}$$

$$\boxed{K = \cdots \mid C\ T_1\ \ldots\ T_k \mid \cdots, \qquad \text{nonlinear } T_i}$$

$$K\mathcal{I}\ \frac{\begin{array}{c} A_1 \vdash t_1 : T_1 \\ \cdots \\ A_k \vdash t_k : T_k \end{array}}{A_1, \ldots, A_k \vdash (C\ t_1\ \ldots\ t_k) : K}$$

$$\boxed{K = C_1\ T_{11}\ \ldots\ T_{1k_1} \mid \cdots \mid C_n\ T_{n1}\ \ldots\ T_{nk_n}, \qquad \text{nonlinear } T_{ij}}$$

$$K\mathcal{E}\ \frac{\begin{array}{c} A \vdash u : K \\ B, x_{11} : T_{11}, \ldots, x_{1k_1} : T_{1k_1} \vdash v_1 : V \\ \cdots \\ B, x_{n1} : T_{n1}, \ldots, x_{nk_n} : T_{nk_n} \vdash v_n : V \end{array}}{A, B \vdash (\textbf{case}\ u\ \textbf{of}\ C_1\ x_{11}\ \ldots\ x_{1k_1} \to v_1 \mid \cdots \mid C_n\ x_{n1}\ \ldots\ x_{nk_n} \to v_n) : V}\ x_{ij} \notin B$$

$$\textsc{fix}\ \frac{A \vdash t : T \to T}{A \vdash (\mathit{fix}\ t) : T}$$

Non Linear Types の使い方

以下のルールを導入する
ただし、$!T$ は Non Linear Type を表す

$$\frac{A, x : !T \vdash u : U \quad B, x : T, y : U \vdash v : V}{A, B, x : T \vdash (\mathbf{let!}\ (x)\ y = u\ \mathbf{in}\ v) : V} \quad U \text{ safe for } T$$

Linear Type が write access に対応し、Non Linear Type が read access に対応している

u を評価する際には、x は（同時に）何度でも参照できる
v を評価する際には、x は一度だけ使用できる

まとめ

- Rust の型は、Affine Type Systems に影響を受けている
- Affine Type Systems は、Linear Type Systems に弱化規則を導入した型システムである
- Affine Type Systems は、全ての変数が高々 1 度、使用されることを保証する型システムである
- Linear Type Systems は、全ての変数が必ず 1 度のみ使用されることを保証する型システムである
- Non Linear Type と Linear Type を組み合わせることにより、read/write access を表現できる