rustic

cordx56

Kernel/VM探検隊@関西 12回目

2026-05-30

おま誰

- cordx56、かおるこたゃ、など
  - aka: たゃちゃん、たゃさん、など
  - 56.ax や @cordx56 で検索
- 東京都内の大学で博士課程学生
  - Rustの型システムと開発支援が専門
  - 特に借用検査
- 様子のおかしいRustオタク
  - 個人的にはPythonも推していきたい

Q. なぜRustを使わないのですか？

- 所有権は用途に対して複雑すぎる
- コンパイル先ターゲットがサポートされていない
- 信頼性が十分に確立されていない
- etc...

Q. なぜRustを使わないのですか？

- 所有権は用途に対して複雑すぎる
- コンパイル先ターゲットがサポートされていない
- 信頼性が十分に確立されていない
- etc...

RustOwl

Q. なぜRustを使わないのですか？

- 所有権は用途に対して複雑すぎる
- コンパイル先ターゲットがサポートされていない
- 信頼性が十分に確立されていない
- etc...

rustic: Rust into C compiler

cordx56

Kernel/VM探検隊@関西 12回目

2026-05-30

何したの？

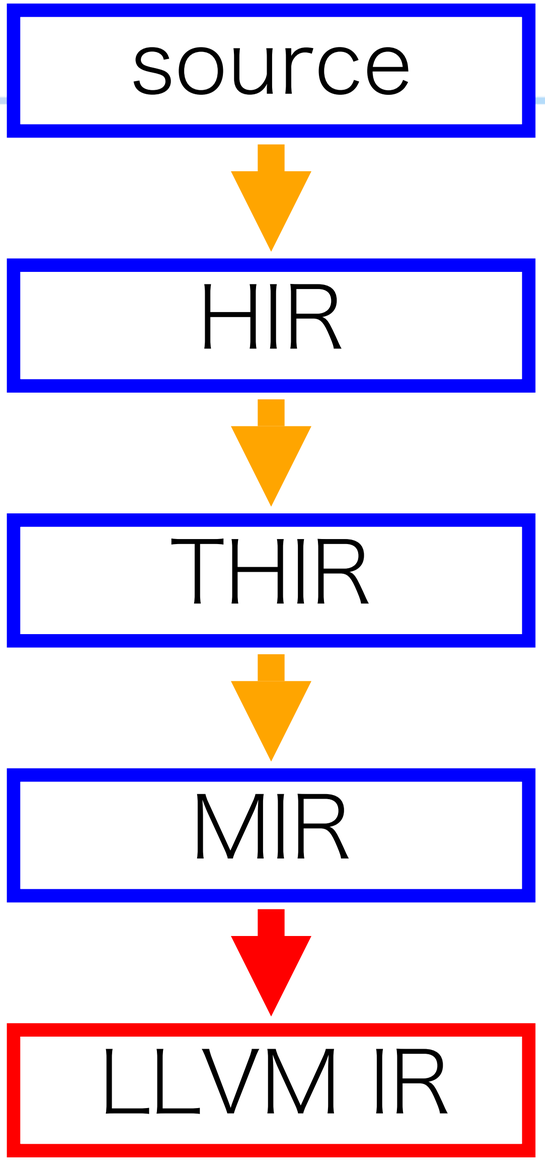
- フルセットRust to Cコンパイラを作った
- Rustコンパイラrustcに追加で実装
- Cコードの吐き出し、コンパイル、リンクを行う
- https://github.com/rustic-compiler
  - 今回は特別に 56.ax/rustic でアクセス可能！

動機

- rustcはLLVMに依存
  - LLVMがサポートしない → ほぼ「サポート外」を意味
- 依然としてC、C++が圧倒的な地位を持つ
- 「Cならいいがは無理」に対応できないか？
  - 実際にはそう簡単に行かない
  - ちなみにとCのFFIという手もある

rustcの内部

- rustcは複数の内部表現を利用
  - HIR: High-level IR
  - THIR: Typed HIR
  - MIR: Mid-level IR
  - LLVM IR: LLVMの中間表現
- MIRからLLVM IRへのコンパイルは `rustc_codegen_llvm` が担当
  - 他には `rustc_codegen_cranelift` など

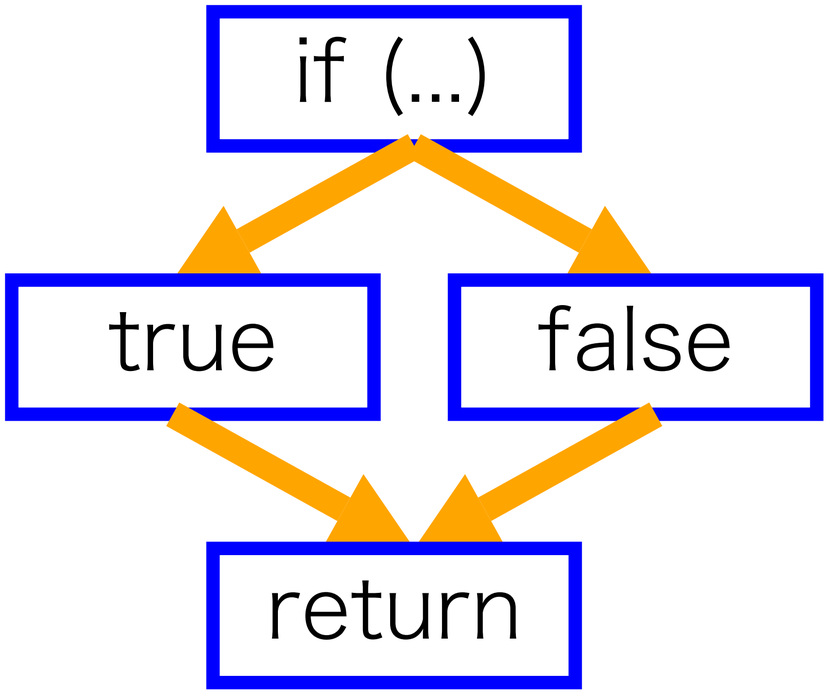


rustc_codegen_c を作る

- rustcの中間表現をCコードへコンパイルするバックエンド
- でもどうやって？

中間表現MIR

- rustc内部で最も低いレイヤーに位置する中間表現
- CFG解析に用いやすい
  - 借用検査もMIRに対して行っている
  - 借用検査についてはまた今度……
- どのような構造になっている？



MIRの構造

- 1つの関数は複数のbasic blockからなる
- 1つのbasic lockは複数のstatementと1つのTerminatorからなる

- 四則演算はstatement
- 参照、dereferenceもstatement
- 関数呼び出しはterminator
- etc...

MIRの性質

- MIRの形式は **Static Single Assignment**（もどき）
  - 日本語だと「静的単一代入」とか
  - LLVM IRも同様の形式
- Terminatorは関数呼び出し、条件分岐などと共に、**次のbasic blockへのジャンプ**を含むことがある
- goto文では……？ :thinking_face:

Cコードの書き出し

1. MIRの各statementをCコードに
   - 四則演算とかref/derefとか
2. TerminatorをCコードに
   - 関数呼び出し、drop、return、etc...

   ここまででブロック1つのコンパイルに成功
3. Terminatorに含まれる次のブロックへの移動をgoto文へ
   - 条件分岐、unwind、etc...

- このようにブロックを順番にCコードへ

モジュールをコンパイル、リンク

- rustcが生成するモジュール単位で `.c` ファイルに
- 最後に `$CC` でコンパイル、リンク
  - 任意のCコンパイラが呼び出せてしまいますねえ〜
- ついでにそのCコードとMakefileを適当なところに配置
  - そこで `make build` すると、Cコードがビルドできるんですね〜

様々なものをコンパイルする

- できたものが妥当かどうか評価するにはテストが必須
- ということで、様々なものを様々な場所でコンパイルしてみよう

Round 1: Hello, world!

- ```
  fn main() {
      println!("Hello, world!");
  }
  ```
- 実行: `cargo +rustic run`
- できて当然
  - gcc 13.3.0, aarch64, Linux 6.17.0上で確認
  - 以降注釈無き場合は同じ環境でテスト
  ```

Round 2: 他の `std` を使うコード

- ```
  fn main() {
      let v = vec![1, 2, 3];
      println!("{v:?}");
  }
  ```
- 出力: `[1, 2, 3]`
- これも難なく正しい出力を得られた
  ```

Round 3: その辺のクレートを片っ端から

- パッケージレジストリ `crates.io` からダウンロード数上位のクレートをいくつか落としてきてコンパイルするPythonスクリプトを実行

- 以下のクレートでコンパイルに成功（上位20件全て通過）
  - `syn @2.0.117`
  - `hashbrown @0.17.1`
  - `bitflags @2.11.1`
  - `getrandom @0.4.2`
  - `rand_core @0.10.1`
  - `libc @0.2.186`
  - `proc-macro2 @1.0.106`
  - `base64 @0.22.1`

（左は意図的に溢れさせています）

Round 4: rustc、std自身

- rustcはRustで書かれている
- rustcのコンパイルは複数ステージに分けて行われる
  - stage 0: コンパイル済みのものをダウンロード
  - stage 1: stage 0コンパイラでビルド
  - stage 2: stage 1コンパイラでビルド
- stage 2までビルド → セルフホスト成功
- 無事成功

Round 5: rustcのUIテスト

- 実際のrustcのテストに用いられているUIテスト
  - 変更されたrustcが正しくエラー出力、コンパイルできるかテスト
- 20,408件中99.2%を通過
  - 通過しなかったのは主にSIMDなどのアーキ依存関連

Round 6: 他のアーキ、CCではどうか

- 様々なアーキ上で試したい
  - 一方手元にあったのはaarch64サーバとamd64ノートPCのみ……
- 他のCコンパイラでもいけるのか？
  - GCC拡張でゴリ押しはつまらないですからね
- とりあえずGitHubでCIを構築
  （rustic-compiler/rustc_codegen_c/.github/workflows/ci.yaml）
  - 前述UIテストを99%以上通過しない場合はエラーになるCI
  - amd64, aarch64, GCC, Clangのmatrix
  - 全て通過

Q. 本当にCコードだけになるんですか？

- 前述の通り、`.c` ファイルとMakefileを書き出す
- `make build` でrustcをビルドすることに成功
  - ビルド時間は長い
- 当然生成された `.c` ファイルは人間可読では無い

Q. コードの量は膨大ではありませんか

- メインの `rustc_codegen_c` は総行数8,854
  - ちょっとだけいじったrustcのビルドスクリプトは除外
- 近年はLLMってやつがめっちゃ書いてくれる
  - ~~AIにパワハラを行うことで全てを解決~~
  - 実際のところはかなり私が方針を示す必要がありました

Q. とはいえそんなに簡単なことですか？

- 様々なworkaroundで誤魔化しており……
  - 特にundefined behavior周り
- 例としては
  - unwind（例外的なやつ）は `setjmp.h` で苦しみながら実現
  - オーバーフローとかの細かい挙動はCのglue codeで関数呼び出し

まとめ

- Rust into Cコンパイラツールチェイン「rustic」開発
- RustをCにコンパイルすることで、ターゲット拡大？
- rustcセルフホスト、複数アーキ、複数コンパイラで検証
- 試してみてね: https://github.com/rustic-compiler
  - なんと今なら 56.ax/rustic でアクセス可能！
  - 私が持ってるドメインなので怪しくないです