CとRで構成された科学ソフトウェアのRustによる再実装と評価

信州大学工学部電子情報システム工学科4年
22T2040H　加井雄一朗

二言語問題 [1], [2], [3]

科学ソフトウェア: 科学分野で数値計算や解析などに使われるソフトウェア。

科学ソフトウェアでは、プロトタイプを生産性の高い言語で開発した後、性能のために部分的にパフォーマンスが高い言語で書き直すことがある。開発・拡張する際に、2つの異なるプログラミング言語を使わざるを得ない状況を示し、保守の面で問題が生じる場合がある。
例）PythonとC++、RとC

最初からJuliaやRustなどのある程度の生産性とパフォーマンスを担保する言語で開発することも進められている。しかし、既存の科学ソフトウェアなどにおいては依然として二言語問題は存在する。

[1]Jaemin Hong, Sukyoung RyuJeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah. (2014). “Julia: A Fresh Approach to Numerical Computing”. https://doi.org/10.48550/arXiv.1411.1607
[2]Johannes Köster. (2016). “Rust-Bio: a fast and safe bioinformatics library”. https://doi.org/10.1093/bioinformatics/btv573
[3]Jeffrey M. Perkel. (2020). “Why scientists are turning to Rust”. https://doi.org/10.1038/d41586-020-03382-2

TRACTOR: Translating All C to Rust [4]

DARPA（米国防総省国防高等研究計画局）が進めるプログラム:

C/C++は20年以上にわたりメモリ安全性の問題を抱えている。バグ検出ツールだけでは不十分。
⇨　根本的な解決には安全なプログラミング言語が必要

既存のCコードをメモリ安全性に優れたRustに自動変換することを目的とする。熟練のRust開発者が生み出すのと同じ品質とスタイルを実現し、Cプログラムに存在する**メモリ安全性の脆弱性を排除する**ことを目標とする。

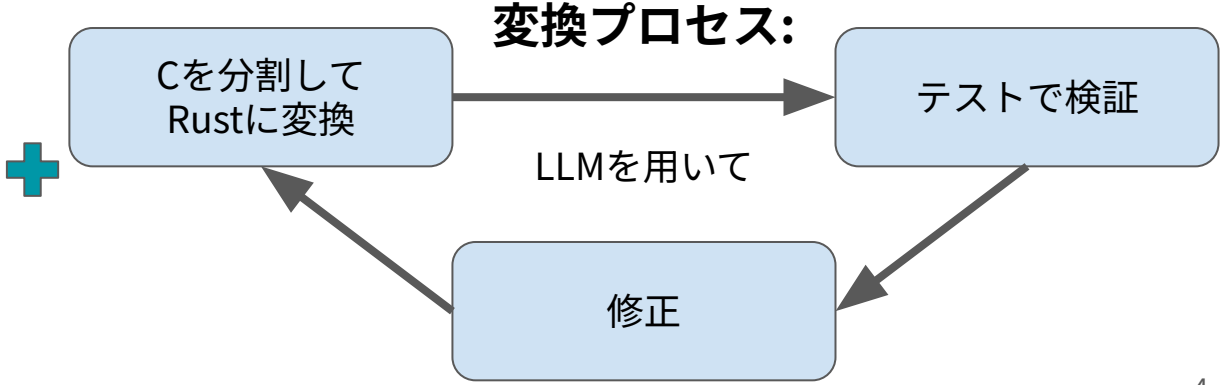
[4]DARPA. (2024). “TRACTOR: Translating All C to Rust”. https://www.darpa.mil/research/programs/translating-all-c-to-rust

先行研究: CからRustへの自動変換 [5]

C2Rustなどの自動変換ツールは**Rustらしくない**コードを生成してしまう。

- Unsafeな機能の残存　⇒　Rustコンパイラが安全性の保証ができないコード
- 所有権の欠落　⇒　メモリに関するエラーの原因になるコード
- 非イディオム的な表現　⇒　C特有の出力パラメータなどが残るコード

静的解析:
コードを実行せずに、コード自体を分析して問題を開発の早い段階で検出する手法

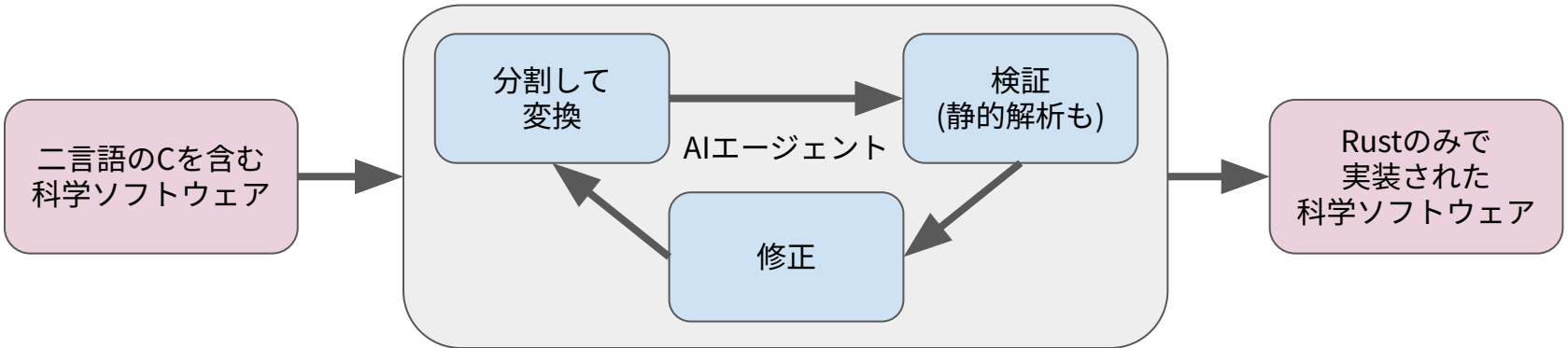


[5]Jaemin Hong, Sukyoung Ryu. (2025). “Automatically Translating C to Rust”. https://cacm.acm.org/research/automatically-translating-c-to-rust/

提案するアプローチ

既存のCを含む科学ソフトウェアの二言語問題を解決するために

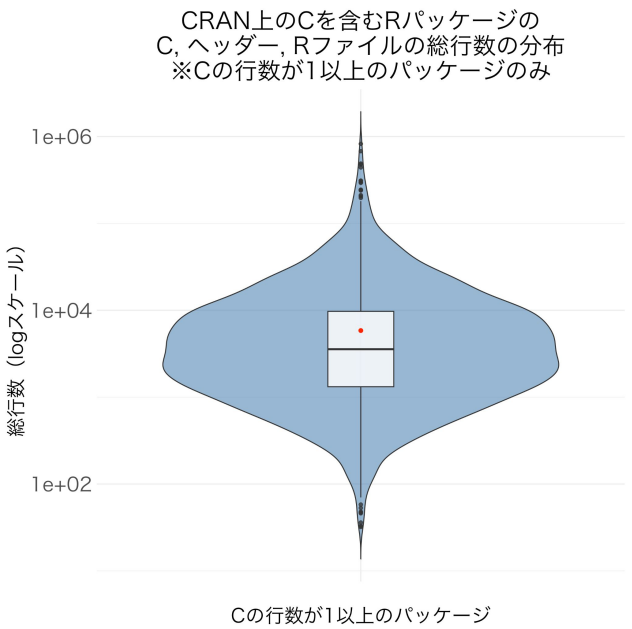
- 二言語のCを含む科学ソフトウェアをRustのみで実装された科学ソフトウェアに変換する。
  ⇨　Rustのみで維持管理を可能にして**保守性を改善**
- 静的解析と共にAIエージェントで「分割して変換→検証→修正→」の変換プロセスを繰り返し進める。
  ⇨　**メモリ安全性が高い**Rustらしいコードに



評価実験の設定

変換対象とする科学ソフトウェア: stringdist
使用されている言語: R, C
変換先の言語: Rust
使用するAIエージェント: GitHub Copilot（Visual Studio Code上）
使用するモデル: GPT-5.2-Codex

stringdist: データサイエンス、特にデータクレンジングや自然言語処理の初期段階で頻繁に使用される。10種類のコアアルゴリズムを実装。

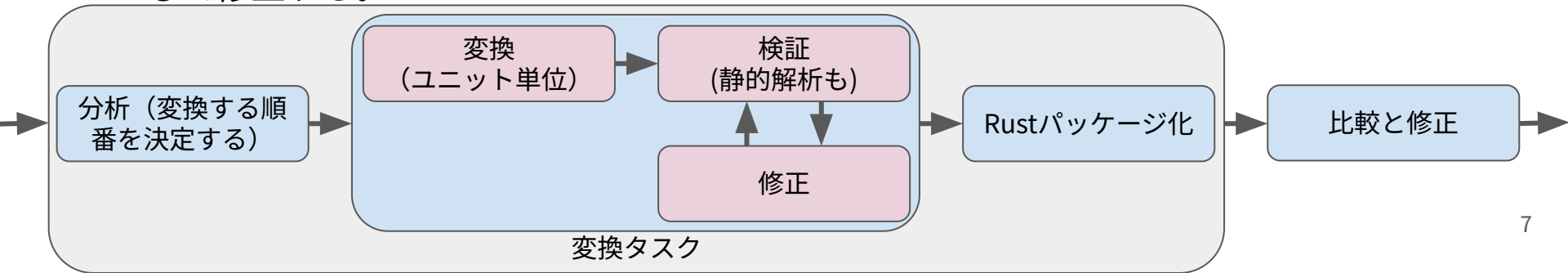


※CRAN: Rのパッケージを保管するR公式のネットワーク

AIエージェントの変換タスク

stringdistをstringdist-rustに変換する。

- 変換タスク全体を3つのフェーズに分割し、プロンプト（AGENTS.md）に変換タスク全体と各フェーズの内容やルールなどを記述し、AIエージェントに与える。
  ※**Rustらしさ（メモリ安全性の高さ、明示的な所有権と型、慣用的なエラー処理）[5]と静的解析ツール（cargo geiger）**でUnsafeを調査することを記述する。
- プロンプトでトリガーを設定し、1回の指示で3つのフェーズに分割した変換タスク全体を完遂させる。変換された後のソフトウェアとＲでの実行結果を比較し、それに応じて修正する。



評価

1. **正確性**
   単体テスト、統合テストが全部合格するか。
   ※統合テストでR版での実行結果との差分検証も行う。
2. **速度**
   生成したデータセットをコアアルゴリズム毎に計測し、中央値で比較する。
3. **保守性**
   静的解析ツール（lizard）を用いて、NLOC（コメントと空行を除いた行数）、CCN（循環的複雑度: 線形的に独立している経路の数）を計算し比較する。
4. **メモリ安全性**
   静的解析ツール（cargo geiger）を用いてRust版のUnsafeの割合を計算する。

正確性の評価結果

- 単体テスト（アルゴリズム内部の分岐、境界条件、例外の挙動を確認）: 22件
  統合テスト（公開APIの入出力、API同士の整合性を確認と差分検証）: 42件
  ※R版での実行結果との差分検証: 5,100ケース（各機能の各メソッドにつき100ケースをランダムに生成し、CSVに保存したデータセット）
  ⇨　**全てのテストに合格**。

- テストカバレッジ測定ツール（cargo tarpaulin）を用いた**テストカバレッジ**の計測では**91.90%**。

速度の評価結果

コアアルゴリズム	R版（ms）	Rust版（ms）	**比率（Rust/R）**
Cosine	14.915223	1.395174	**0.093540**
Jaccard	12.019031	1.492381	**0.124168**
Q-gram	12.019031	1.549248	**0.124200**
Soundex	0.375121	0.077505	**0.206613**
Levenshtein	0.567162	0.345932	**0.609935**
Hamming	0.080900	0.051962	**0.642299**
Damerau	0.727720	0.577895	**0.794117**
LCS	0.214770	0.236247	**1.100000**
OSA	0.380042	0.432818	**1.138869**
Jaro-Winkler	0.135071	0.161693	**1.197096**

※長さ10の英数字文字列を1,000組生成し、CSVに保存したデータセットを用いた。
　R版、Rust版での実行時間は、データセットを100回計測した中央値で比較した。

比率の中央値は0.63となった。
R版での実行時間のオーダーが他とは違っていた3種類のコアアルゴリズムの比率が全て0.15未満となった。

保守性・メモリ安全性の評価結果

指標	R版（実行部）	Rust版（実行部）
NLOC（行）	2,108	1,632
CCNの中央値	3	3

- Rust版では、R版と比べて**NLOCを78%まで削減した。**
  また、lizardではCCNが16以上となる関数に警告が出される。
  R版では警告が合計5件（Rコードで1件、Cコードで4件）。Rust版では警告が0件。

- Unsafeなコードは0%であり、**すべてSafeなコード**で実装された。

考察

- 全てのテストを合格
  比率（Rust版での実行時間/R版での実行時間）が中央値0.63
  Rust版でNLOCを78%、CCNの警告を5件から0件に削減
  ⇨ **同じ動作、より高い性能・保守性**

- Rust版でNLOCを78%に削減
  ⇨ Rust版でFFI（Foreign Function Interface）が不要になったため
  単言語ソフトウェアの利点
  ※FFI: あるプログラミング言語から他のプログラミング言語で定義された関数などを利用するための機構。

考察

- すべてSafeなコード
  ⇨　**メモリ安全性**が高いRustらしいコード

- 比率が0.15未満のR版の3種類のコアアルゴリズムでは**二分木構造**を使っている（他のコアアルゴリズムでは二分木構造は使っていない）。Rust版では二分木構造の代わりに、**HashMap**を使っている。
  ⇨　二分木構造とHashMapの挿入、参照、削除の時間計算量はそれぞれ対数時間と定数時間であるため、このアルゴリズムの差が性能改善に貢献していると思われる。

まとめ

- stringdistをRustに変換することにより、**保守性**を改善することができた。

- 静的解析と共にAIエージェントで「分割して変換→検証→修正→」の変換プロセスを繰り返し進めることで、**メモリ安全性**が高いRustらしいコードとして変換することができた。

- stringdistをstringdist-rustに変換することは可能で、既存のCを含む科学ソフトウェアの**二言語問題**を解決できる可能性を示した。