# Eine Fallstudie zur manuellen Transformation des QOI-Bildformats von automatisch generiertem C2Rust-Code zu idiomatischem Rust.

Stöcklein, Daniel
*Hochschule Mannheim*
*Fakultät für Informatik*
*Paul-Wittsack-Str. 10, 68163 Mannheim*

***Zusammenfassung*—**

## Inhaltsverzeichnis



# 1. Einleitung

## 1.1. Motivation

## 1.2. Aufgabenstellung und Zielsetzung

# 2. Grundlagen und Definitionen

## 2.1. Automatisierte Transpilierung mit C2Rust

## 2.2. Speicherverwaltung: Manuelle Ressourcenverwaltung (C) vs. Ownership-Modell

## 2.3. Das QOI (Quite OK Image) Format

# 3. Problem- und Anforderungsanalyse

## 3.1. Analyse des Ist-Zustands (C2Rust Output)

## 3.2. Idiomatischer Rust Code

# 4. Konzept und Lösungsansatz

## 4.1. Unsicherheits-Muster und Ableitung von Transformationsregeln

Die automatisierte Transpilierung mittels C2Rust liefert ein Ergebnis, das zwar funktional äquivalent, aber strukturell unsicher ist. Um diesen Code in idiomatisches Rust zu überführen, werden im Folgenden generische Transformationsregeln abgeleitet, die später auf den C2Rust Output von QOI angewandt [1].

**4.1.1. Transformation von Zeigerarithmetik in Slice-Operationen.** Das gravierendste Sicherheitsrisiko im generierten Code ist die direkte Manipulation von Rohzeigern (*mut u8). C-Code verlässt sich hierbei auf implizite Invarianten (zum Beispiel, dass der Puffer groß genug ist). Rust hingegen erzwingt durch das Ownership-Modell explizite Grenzen [2]. Als zentrale Maßnahme wird daher die **Überführung unsicherer Rohzeiger-Manipulationen in grenzgeprüfte Slice-Operationen** definiert, um deterministische Speichersicherheit zu gewährleisten.

**4.1.2. Automatisierung der Ressourcenverwaltung.** C-Programme delegieren die Verantwortung für die Freigabe von Speichern mittels `free()` vollständig an den Entwickler. C2Rust behält diese manuellen Aufrufe bei und markiert sie als **unsafe**. Dies birgt die Gefahr von Memory Leaks oder Double-Free-Fehlern [3]. Daher ist es sinnvoll, manuelle Allokationen (`libc::malloc`) durch Rust-Standardtypen wie `Vec<u8>` oder `Box<T>` zu ersetzen. Durch den in Rust implementierten Drop-Trait wird sichergestellt, dass Speicher automatisch freigegeben wird [4]. Die hieraus abgeleitete Transformationsregel fordert den **Ersatz manueller Allokationsroutinen durch RAII-konforme Standardtypen**, um die Speicherfreigabe mittels des Drop-Traits fest an den Gültigkeitsbereich zu binden.

**4.1.3. Kapselung von globalem Zustand (Mutable Statics).** Der QOI-Algorithmus nutzt in seiner C-Form teilweise statische Arrays oder Puffer, die global sichtbar sind. C2Rust übersetzt diese in static mut Variablen. Veränderbare globale Variablen in Rust sind zwingend unsafe, da sie Data Races in multithreaded Umgebungen ermöglichen [5]. Zur Wiederherstellung der Thread-Sicherheit gilt die Regel der **Elimination global veränderbarer statischer Variablen zugunsten gekapselter Kontext-Strukturen**.

**4.1.4. Transformation von imperativen Schleifen in funktionale Iteratoren.** Der C2Rust-Transpiler übersetzt C-Schleifen (for, while) oft in primitive loop-Konstrukte mit manuellen break-Anweisungen und Index-Variablen. Dies führt zu einem imperativen, schwer lesbaren Kontrollfluss. Modernes Software-Engineering favorisiert deklarative Ansätze wie `.iter()` oder `.map()` [6]. Diese Iteratoren reduzieren „Off-by-one“-Fehler, da Zugriffe nicht mehr durch Indizes erfolgen. Dies führt zur Transformationsregel, **imperative Schleifenkonstrukte in deklarative Iterator-Ketten zu überführen**, um Indexierungsfehler zu reduzieren und die semantische Lesbarkeit zu erhöhen.

**4.1.5. Typsichere Fehlerbehandlung.** C nutzt häufig Rückgabewerte wie 0 oder -1, um Erfolg oder Misserfolg zu signalisieren. Dieser Ansatz ist semantisch mehrdeutig und zwingt den Aufrufer, die Dokumentation zu konsultieren. Idiomatisches Rust nutzt das `Result<T, E>`-Enum, um Fehler im Typsystem unumgänglich zu machen. Daraus resultiert die **Ablösung semantisch ambivalenter Rückgabewerte durch das `Result<T, E>`-Pattern** zur Erzwingung einer expliziten Fehlerbehandlung.

Durch diese Kategorisierung wird der Refactoring-Prozess von einer subjektiven Stil-Korrektur zu einem deterministischen Verfahren. Jeder im C2Rust-Output gefundene Verstoß gegen diese drei Kategorien wird gemäß der definierten Regel transformiert.

### 4.2. Methodik der Verifikation und Definition von Qualitätsmetriken

Da die zuvor hergeleitete Refactoring-Strategie tiefgreifende Eingriffe in die Speicherverwaltung des QOI-Algorithmus vorsieht, ist ein Validierungskonzept unerlässlich. Die bloße Kompilierbarkeit des Rust-Codes ist kein hinreichender Beweis für die semantische Korrektheit. Daher stützt sich diese Arbeit auf zwei Evaluationsmodelle: Funktionale Äquivalenz und Code-Qualität.

**4.2.1. Gewährleistung der funktionalen Äquivalenz durch Differential Testing.** Risiko einer manuellen Migration ist die unbemerkte Einführung von Regressionen. Um dies auszuschließen, wird der Ansatz des Differential Testing verfolgt [7]. Hierbei werden die originale C-Implementierung und das Rust-Refactoring als ”Black Boxes”betrachtet, die bei identischem Input ($I$) zwingend identischen Output ($O$) liefern müssen ($O_C \cdot (I) = O_{Rust} \cdot (I)$). Für das QOI-Format bedeutet dies, dass ein Testkorpus aus zufälligen Bilddaten (Fuzzing) sowie definierten Randfällen (zum Beispiel Bilder mit Breite 0 oder extremen Seitenverhältnissen) erstellt wird. Abweichungen im Byte-Stream des encodierten Bildes werden als Fehlfunktion gewertet. Dieser Ansatz eliminiert die subjektive Fehleranfälligkeit des Entwicklers beim Code-Review.

**4.2.2. Statische Analyse und Komplexitätsmetriken.** Das Ziel der Arbeit ist nicht nur die funktionale Korrektheit, sondern die Erhöhung der Wartbarkeit und Sicherheit. Um diesen qualitativen Fortschritt quantifizierbar zu machen, wird die Zyklomatische Komplexität betrachtet [8]. C-Code, der Zeigerarithmetik zur Iteration nutzt, weist oft eine implizite Komplexität auf, die schwer zu erfassen ist. Zusätzlich dient der Rust-Linter Clippy als objektive Instanz zur Bewertung der „Idiomatik“. Während der C2Rust-Output hunderte von Warnungen generiert, dient die Reduktion dieser Linter-Warnungen gegen Null als messbarer Indikator für die Annäherung an idiomatischen Rust-Code.

## 5. Implementierung

### 5.1. Aufbau der Entwicklungsumgebung

### 5.2. Restrukturierung der Datenstrukturen

### 5.3. Refactoring der Kernlogik (Encoder/Decoder)

### 5.4. Behandlung von Randfällen und Fehlerzuständen

### 5.5. Testen auf Korrektheit des transformierten Rust Codes

## 6. Validierung und Nachweis der Richtigkeit

### 6.1. Ergebnisse des Differential Testing

### 6.2. Analyse der Code-Qualität (Metriken, Linter-Ergebnisse)

## 7. Zusammenfassung und Ausblick

## Literatur


[1] G. Lacerda, F. Petrillo, M. Pimenta und Y. G. Guéhéneuc, „Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations“, *Journal of Systems and Software*, Jg. 167, S. 110 610, 1. Sep. 2020. DOI: 10.1016/j.jss.2020.110610 besucht am 27. Nov. 2025. Adresse: https://www.sciencedirect.com/science/article/pii/S0164121220300881

[2] H. Zhang, C. David, Y. Yu und M. Wang. „Ownership Guided C to Rust Translation“. arXiv: 2303.10515 [cs], besucht am 5. Okt. 2025. Adresse: http://arxiv.org/abs/2303.10515 Vorveröffentlichung.

[3] B. Stroustrup, *The C++ Programming Language*, 4. Aufl. Boston, MA: Addison-Wesley Professional, 2013.

[4] B. Xu, B. Chu, H. Fan und Y. Feng, „An Analysis of the Rust Programming Practice for Memory Safety Assurance“, in *Web Information Systems and Applications*, L. Yuan, S. Yang, R. Li, E. Kanoulas und X. Zhao, Hrsg., Singapore: Springer Nature, 2023, S. 440–451. DOI: 10.1007/978-981-99-6222-8_37

[5] M. Emre, R. Schroeder, K. Dewey und B. Hardekopf, „Translating C to safer Rust“, *Proceedings of the ACM on Programming Languages*, Jg. 5, S. 1–29, OOPSLA 20. Okt. 2021. DOI: 10.1145/3485498 besucht am 5. Okt. 2025. Adresse: https://dl.acm.org/doi/10.1145/3485498

[6] J. Blandy, J. Orendorff und L. F. Tindall, *Programming Rust*. ” O’Reilly Media, Inc.”, 2021.

[7] M. A. Gulzar, Y. Zhu und X. Han, „Perception and Practices of Differential Testing“, in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Mai 2019, S. 71–80. DOI: 10.1109/ICSE-SEIP.2019.00016 besucht am 30. Nov. 2025. Adresse: https://ieeexplore.ieee.org/abstract/document/8804465

[8] C. Ebert, J. Cain, G. Antoniol, S. Counsell und P. Laplante, „Cyclomatic Complexity“, *IEEE Software*, Jg. 33, Nr. 6, S. 27–29, Nov. 2016. DOI: 10.1109/MS.2016.147 besucht am 30. Nov. 2025. Adresse: https://ieeexplore.ieee.org/abstract/document/7725232