Projektarbeit Pony

Atay Oezcan und Daniel Weingand

21. Januar 2026

Inhaltsverzeichnis

1 Einleitung

Die Entwicklung nebenläufiger Software stellt Programmierer seit Jahrzehnten vor erhebliche Herausforderungen. Klassische Probleme wie Data Races, Deadlocks und Memory Leaks führen zu schwer reproduzierbaren Fehlern, die oft erst in Produktionsumgebungen auftreten. Diese Projektarbeit untersucht die Programmiersprache Pony und analysiert, wie ihr Typsystem und Aktormodell diese fundamentalen Probleme strukturell löst.

Für die Untersuchung wurden fünf Programmiersprachen ausgewählt: C++ als etablierter Industriestandard, Go als moderne Systemsprache mit integrierter Nebenläufigkeit, Rust mit seinem Ownership-System, Swift mit Actor-Unterstützung seit Version 5.5, sowie Pony als spezialisierte Aktorsprache. Jede Sprache wurde anhand identischer Testszenarien evaluiert, die typische Nebenläufigkeitsprobleme provozieren.

2 Die Programmiersprache Pony

Pony ist eine objektorientierte Open-Source-Programmiersprache, die 2015 an der Imperial College London entwickelt wurde. Die Sprache kombiniert das Aktormodell mit einem neuartigen Typsystem namens Reference Capabilities, das zur Kompilierzeit garantiert, dass keine Data Races auftreten können. Anders als bei vielen modernen Sprachen, die Sicherheit durch Laufzeitprüfungen erreichen, verlagert Pony diese Garantien vollständig in den Compiler.

2.1 Das Aktormodell in Pony

Das Aktormodell bildet das Fundament der Nebenläufigkeit in Pony. Jeder Aktor repräsentiert eine unabhängige Berechnungseinheit mit eigenem Speicherbereich, der von keinem anderen Aktor direkt zugegriffen werden kann. Die Kommunikation erfolgt ausschließlich über asynchrone Nachrichten.

Dieser Ansatz unterscheidet sich grundlegend von threadbasierten Modellen. Während Threads typischerweise Speicher teilen und Synchronisationsmechanismen wie Mutexe benötigen, operieren Aktoren vollständig isoliert. Eine Nachricht an einen Aktor wird in dessen Mailbox eingereiht und sequenziell abgearbeitet. Da der Aktor seinen Zustand nur selbst modifizieren kann und keine externe Synchronisation erforderlich ist, werden Data Races strukturell unmöglich.

Der Nachteil dieses Ansatzes liegt im Speicherverbrauch: Da kein Speicher geteilt wird, müssen Daten zwischen Aktoren kopiert oder als unveränderliche Referenzen übergeben werden. Pony adressiert dies durch Reference Capabilities, die dem Compiler ermöglichen, sichere Referenzübergaben zu optimieren.

2.2 Reference Capabilities

Das Typsystem von Pony erweitert klassische Typen um sogenannte Reference Capabilities. Diese beschreiben, welche Operationen mit einer Referenz erlaubt sind und ob sie mit anderen Aktoren geteilt werden darf. Die wichtigsten Capabilities sind `iso` für isolierte, nicht teilbare Referenzen, `val` für unveränderliche, teilbare Werte, sowie `ref` für veränderliche, nicht teilbare Referenzen.

Der Compiler analysiert zur Kompilierzeit den Datenfluss und stellt sicher, dass niemals zwei Aktoren gleichzeitig schreibenden Zugriff auf denselben Speicher haben. Dieses System ist strenger als das Ownership-System von Rust, da es auch Deadlocks durch sein Design verhindert.

2.3 ORCA Garbage Collection

Die Speicherverwaltung in Pony basiert auf ORCA (Ownership and Reference Counting for Actors), einem speziell für das Aktormodell entwickelten Garbage Collector. Im Gegensatz zu herkömmlichen Garbage Collectoren, die das gesamte Programm anhalten müssen, arbeitet ORCA nebenläufig auf Aktor-Ebene.

Jeder Aktor führt seine eigene Garbage Collection durch, ohne andere Aktoren zu blockieren. Dies ermöglicht vorhersagbare Laufzeiten und vermeidet die bei Go und Java bekannten GC-Pausen. Zudem werden Aktoren, die keine Nachrichten mehr empfangen können, automatisch aufgeräumt, was Goroutine-Leaks wie in Go strukturell verhindert.

2.4 Einschränkungen

Trotz dieser Vorteile hat Pony als junge Sprache praktische Einschränkungen. Die Sprache hat Version 1.0 noch nicht erreicht, was zu gelegentlichen API-Änderungen führt. Das Ökosystem ist überschaubar, mit Bibliotheken hauptsächlich für HTTP, JSON, Datenbanken und Netzwerkprogrammierung. Entwicklungsumgebungen bieten keine native Unterstützung, und Analyse-Tools sind kaum verfügbar.

3 Vergleich mit Swift Aktoren

Swift führte mit Version 5.5 ein eigenes Aktormodell ein, das sich in mehreren Aspekten von Ponys Ansatz unterscheidet. Beide Sprachen verwenden Aktoren als Grundlage für sichere Nebenläufigkeit, verfolgen jedoch unterschiedliche Philosophien.

In Pony sind Daten grundsätzlich unveränderlich, wenn sie zwischen Aktoren geteilt werden. Swift hingegen erlaubt veränderliche Daten innerhalb

eines Aktors, wobei das System sicherstellt, dass immer nur ein Thread diese Daten modifizieren kann. Die Kommunikation in Pony erfolgt über ein explizites Nachrichtensystem, während Swift asynchrone Methodenaufrufe verwendet, die syntaktisch näher an gewöhnlichen Funktionsaufrufen liegen.

Ein wesentlicher Unterschied besteht in der Fehlertoleranz. Ponys strikte Isolation der Aktoren ermöglicht hohe Fehlertoleranz, da ein fehlgeschlagener Aktor andere nicht beeinträchtigt. Swift bietet zwar sicheren Zugriff auf Aktor-Zustände, hat aber keine vergleichbare Isolation. Bei der Skalierbarkeit profitiert Pony von seiner effizienten Nachrichtenzustellung, während Swift auf optimierten Thread-Zugriff setzt.

4 Experimentelle Untersuchung

Für die praktische Evaluation wurden fünf Testszenarien implementiert, die jeweils ein spezifisches Nebenläufigkeitsproblem adressieren. Alle Tests wurden in allen fünf Sprachen umgesetzt, um direkten Vergleich zu ermöglichen.

4.1 Data Races

Der erste Test provoziert Data Races durch zwei Threads, die gleichzeitig einen gemeinsamen Zähler inkrementieren. Jeder Thread führt 100.000 Inkrementierungen durch, das erwartete Ergebnis beträgt 200.000.

In C++ und Go tritt das Race auf und führt zu unvorhersehbaren Ergebnissen unter 200.000. Go bietet einen Race Detector, der das Problem zur Laufzeit erkennt, verhindert es aber nicht. Swift warnt seit Version 6 vor unsicherem Code, erlaubt ihn jedoch weiterhin.

Rust verhindert das Data Race zur Kompilierzeit durch sein Ownership-System. Der Compiler weigert sich, Code zu übersetzen, der unsynchronisierten Zugriff auf gemeinsame Daten erlaubt. Pony geht einen Schritt weiter: Da Aktoren keinen Speicher teilen können, ist ein Data Race syntaktisch nicht formulierbar. Der Versuch, einen solchen Test zu schreiben, scheitert bereits am Typsystem.

4.2 Deadlocks

Der Deadlock-Test verwendet zwei Threads, die zwei Mutexe in umgekehrter Reihenfolge anfordern. Thread 1 sperrt Mutex A und wartet auf B, Thread 2 sperrt B und wartet auf A. Diese zirkuläre Abhängigkeit führt zum klassischen Deadlock.

C++, Go, Rust und Swift zeigen alle das gleiche Verhalten: Das Programm hängt und muss extern beendet werden. Bemerkenswert ist, dass selbst Rust, das Data Races verhindert, Deadlocks nicht ausschließen kann. Das Ownership-System garantiert Speichersicherheit, nicht aber Fortschritt.

Pony verhält sich fundamental anders. Da das Aktormodell keine Locks verwendet, kann der Test nicht in der klassischen Form implementiert werden. Die äquivalente Implementierung mit Aktoren und Nachrichten terminiert erfolgreich, da asynchrone Kommunikation keine Blockierung verursacht.

4.3 Dining Philosophers

Das klassische Philosophenproblem veranschaulicht Deadlocks an einem konkreten Szenario. Fünf Philosophen sitzen an einem runden Tisch, zwischen ihnen liegen fünf Gabeln. Zum Essen benötigt jeder Philosoph zwei Gabeln. Wenn alle gleichzeitig ihre linke Gabel aufnehmen, entsteht ein Deadlock.

Die Lösungen in den verschiedenen Sprachen zeigen unterschiedliche Ansätze. C++ und Rust verwenden Mutexe mit asymmetrischer Ressourcenordnung, wobei ein Philosoph die Gabeln in umgekehrter Reihenfolge aufnimmt. Go setzt auf Channels statt geteiltem Speicher. Swift und Pony nutzen Aktoren, wobei jede Gabel ein eigenständiger Aktor ist, der Anfragen sequentiell bearbeitet.

4.4 Memory Leaks

Der Memory-Leak-Test alloziert wiederholt Speicher ohne ihn freizugeben. In C++ führt dies zu klassischen Leaks durch vergessene delete-Aufrufe. Go kann trotz Garbage Collector Leaks haben, etwa durch Goroutinen, die auf Channels warten, die nie beschrieben werden.

Rust verhindert die meisten Leaks durch sein Ownership-System, erlaubt aber Referenzzyklen mit `Rc` und `RefCell`. Swift hat ähnliche Einschränkungen bei Retain Cycles in ARC.

Pony verwendet ORCA, das speziell für das Aktormodell entwickelt wurde. Da Aktoren isoliert arbeiten und automatisch aufgeräumt werden, wenn sie keine Nachrichten mehr empfangen können, sind sowohl klassische Memory Leaks als auch Goroutine-Leaks ausgeschlossen.

4.5 Runtime Errors

Der letzte Test untersucht die Behandlung von Array-Zugriffen außerhalb der Grenzen. C++ zeigt undefiniertes Verhalten, Go und Rust beenden das Programm mit Panic, Swift mit Fatal Error.

Pony ist die einzige Sprache, bei der Array-Zugriffe syntaktisch als potenziell fehlschlagend markiert werden müssen. Der Operator `arr(i)?` erzwingt Fehlerbehandlung, sonst kompiliert das Programm nicht. Diese strenge Anforderung eliminiert eine ganze Klasse von Laufzeitfehlern.

5 Leistungsvergleich

Um zu untersuchen, ob die Sicherheitsgarantien von Pony mit Leistungseinbußen verbunden sind, wurde ein CPU-intensiver Benchmark durchgeführt. Der Sieve of Eratosthenes berechnet alle Primzahlen bis zehn Millionen und summiert sie auf. Dieser Algorithmus testet Speicherallokation, Schleifenoptimierung und reine Rechenleistung.

5.1 Messergebnisse

Mit maximaler Compiler-Optimierung erreichen alle Sprachen vergleichbare Zeiten. Rust und Swift liegen bei etwa 24 Millisekunden, Pony bei 25, Go bei 26 und C++ bei 29 Millisekunden. Die Varianz zwischen den Läufen ist größer als die Unterschiede zwischen den Sprachen.

Die Analyse des generierten Maschinencodes erklärt diese Ähnlichkeit. Alle Compiler erzeugen für die Kernschleife praktisch identischen Assembler: eine Speicheroperation, eine Addition, einen Vergleich und einen bedingten Sprung. Die Unterschiede liegen im Overhead der Laufzeitumgebungen, nicht in der Codequalität.

5.2 Größe der Binärdateien

Deutliche Unterschiede zeigen sich bei der Größe der erzeugten Binärdateien. Swift erzeugt mit 700 Zeilen Assembler die kleinste Datei, da die Foundation dynamisch gelinkt wird. C++ erreicht mit Link-Time Optimization etwa 4.900 Zeilen. Pony liegt bei 23.400 Zeilen, was die statisch eingebundene Aktor-Runtime und ORCA widerspiegelt. Rust und Go erzeugen mit 72.000 bzw. 182.000 Zeilen die größten Binärdateien, da sie ihre kompletten Laufzeitumgebungen statisch linken.

Diese Größenunterschiede beeinflussen die Ausführungsgeschwindigkeit kaum, können aber für eingebettete Systeme relevant sein.

6 Fazit

Die experimentelle Untersuchung zeigt, dass Pony als einzige der getesteten Sprachen sowohl Data Races als auch Deadlocks strukturell verhindert. Rust kommt mit seinem Ownership-System nahe heran, kann aber Deadlocks nicht ausschließen. Swift und Go bieten gute Werkzeuge zur Fehlererkennung, verhindern die Probleme aber nicht grundsätzlich. C++ überlässt die Verantwortung vollständig dem Programmierer.

Die Leistungsmessungen widerlegen die Annahme, dass Sicherheitsgarantien mit signifikanten Performanzkosten verbunden sind. Mit modernen Compilern erreichen alle Sprachen vergleichbare Ausführungszeiten. Die Wahl

der Sprache sollte daher primär auf den Sicherheitsanforderungen und dem verfügbaren Ökosystem basieren.

Für Anwendungen mit hohen Anforderungen an Nebenläufigkeit und Zuverlässigkeit bietet Pony einen einzigartigen Ansatz, der viele klassische Fehlerklassen bereits zur Kompilierzeit ausschließt. Die geringe Verbreitung und das limitierte Ökosystem schränken die praktische Einsetzbarkeit derzeit jedoch ein.

7 Projektstruktur

Das begleitende Repository enthält alle Implementierungen der beschriebenen Tests. Die Ordnerstruktur gliedert sich nach Problemkategorien, wobei jeder Unterordner Implementierungen in allen fünf Sprachen enthält. Die Ordner `data-races`, `deadlocks`, `dining-philosophers`, `memory-leaks` und `runtime-errors` dokumentieren die jeweiligen Probleme und deren sprachspezifische Lösungen.

Der Ordner `leistung` enthält die Benchmark-Implementierungen samt generierter Assembly-Dateien. Die Ordner `server-tcp` und `server-http` zeigen praktische Anwendungen mit Netzwerkkommunikation. Der Ordner `vorstellung` enthält einführende Beispiele zu Ponys Syntax und Konzepten.

Jeder Unterordner ist mit einem README versehen, das die sprachspezifischen Besonderheiten erläutert und Anweisungen zur Ausführung enthält.

8 Quellen

Literatur

[1] Clebsch, Sylvan. *Pony: Co-Designing a Type System and a Runtime*. PhD Thesis, Imperial College London, 2017.

[2] Clebsch, Sylvan; Drossopoulou, Sophia; Blessing, Sebastian; McNeil, Andy. *Deny Capabilities for Safe, Fast Actors*. In: Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!), 2015.