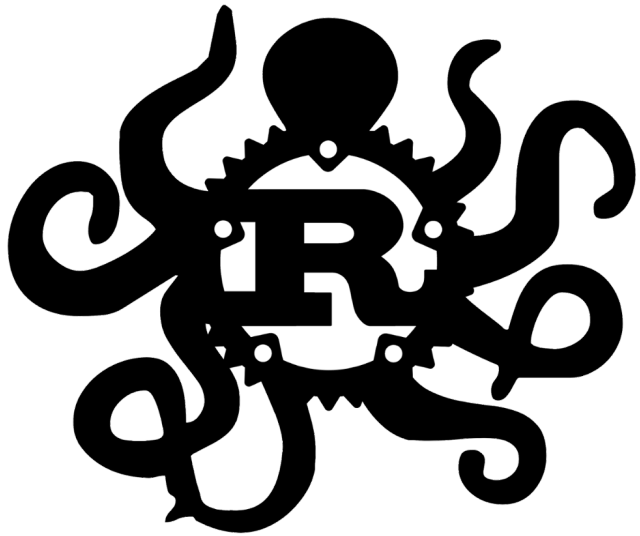
Karlsruher Institut für Technologie

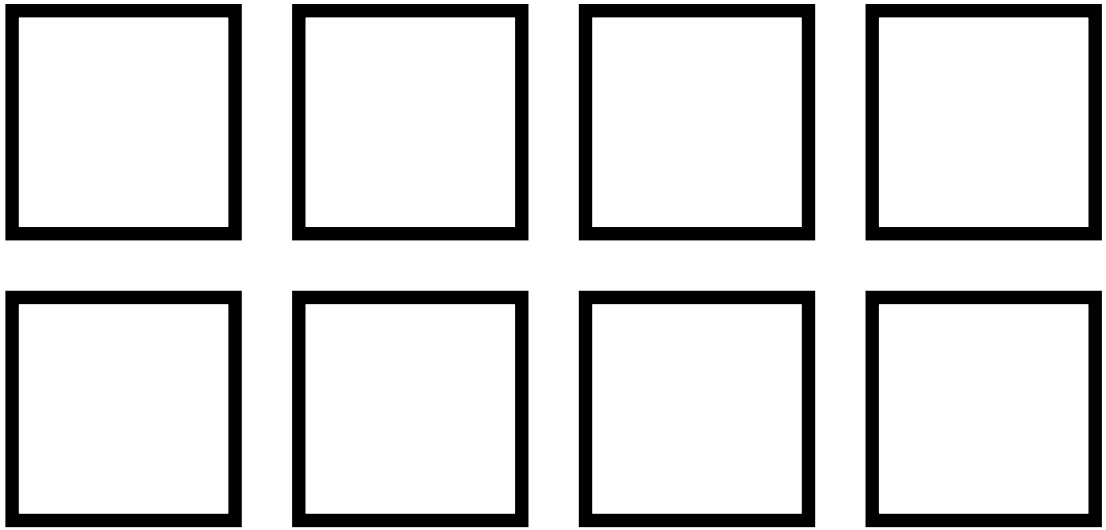
Invasives Rust

Hermann Heinz Erich Krumrey

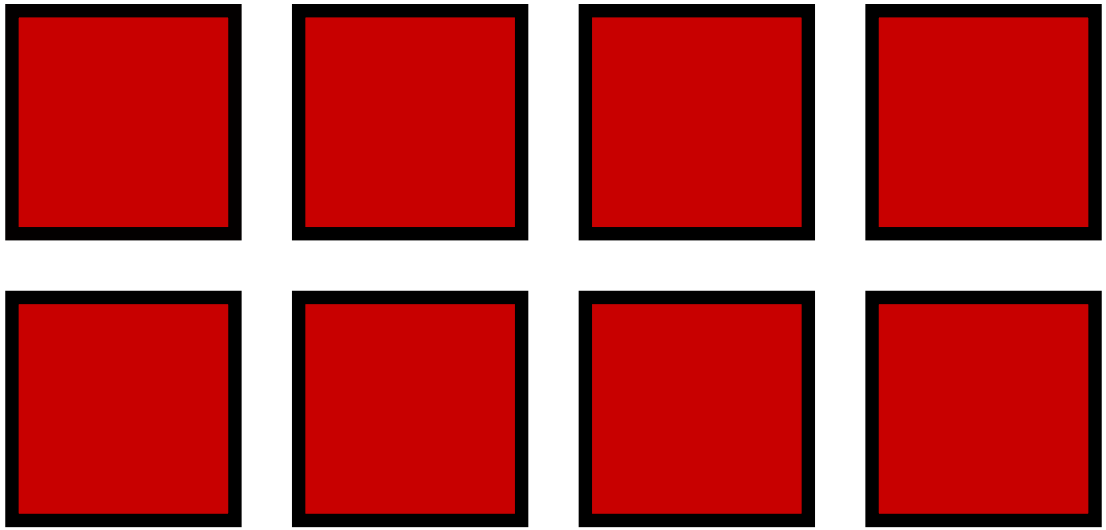
Lehrstuhl Programmierparadigmen, IPD Snelting



Motivation

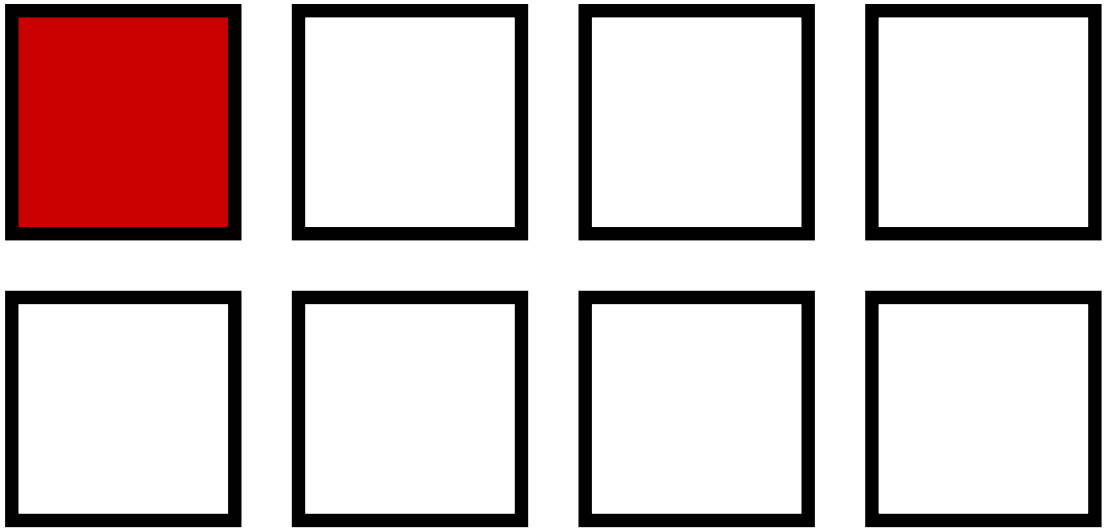


Motivation



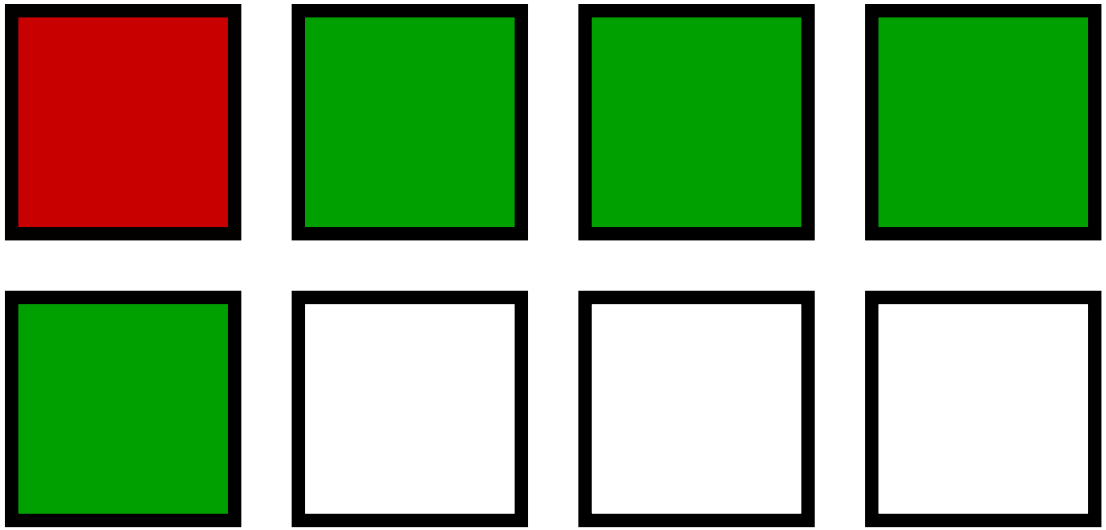
1. Programm 1 beginnt Ausführung auf 8 Recheneinheiten

Motivation



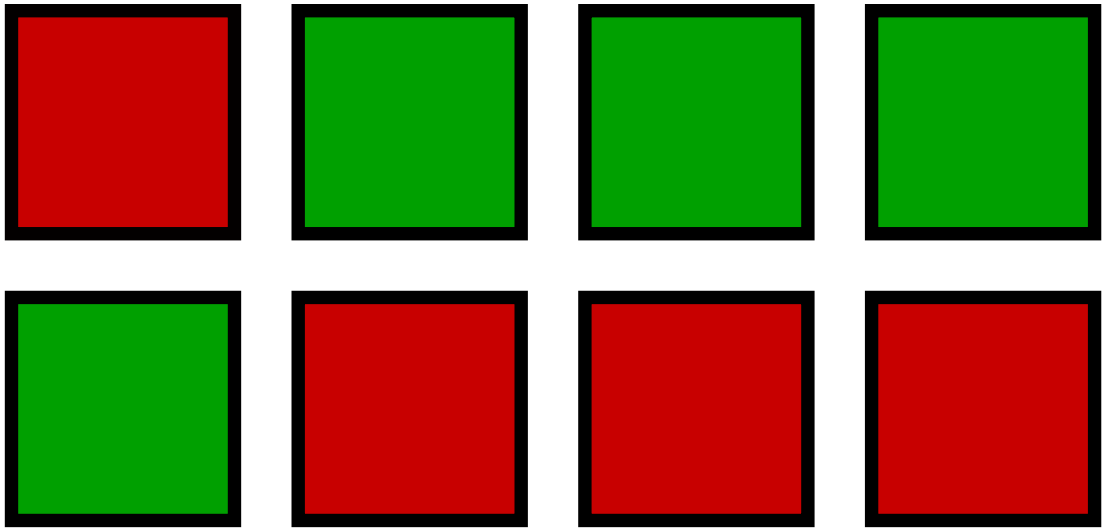
1. Programm 1 beginnt Ausführung auf 8 Recheneinheiten
2. Programm 1 sendet Ergebnisse über das Netzwerk

Motivation



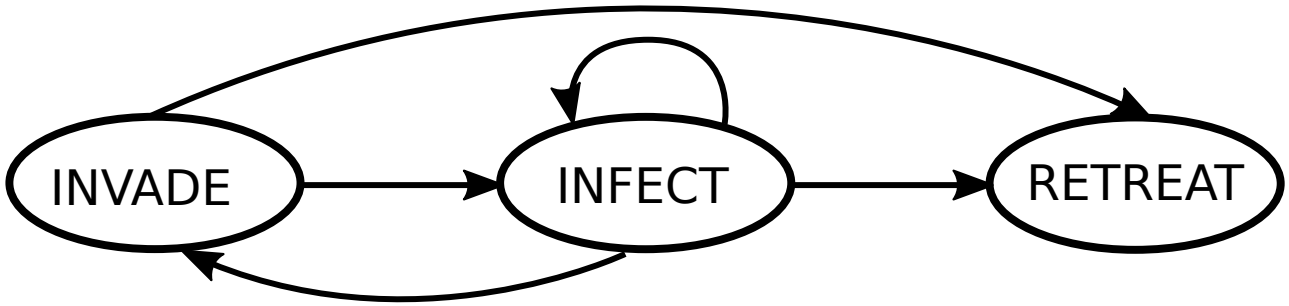
1. Programm 1 beginnt Ausführung auf 8 Recheneinheiten
2. Programm 1 sendet Ergebnisse über das Netzwerk
3. Programm 2 beginnt Ausführung auf 4 Recheneinheiten

Motivation



1. Programm 1 beginnt Ausführung auf 8 Recheneinheiten
2. Programm 1 sendet Ergebnisse über das Netzwerk
3. Programm 2 beginnt Ausführung auf 4 Recheneinheiten
4. Programm 1 führt wieder Berechnungen aus, jetzt auf 4 Recheneinheiten

Invasives Rechnen



- Ressourcenbewusstes Programmieren
- 3 Phasen:
  1. Invade - Ressourcen reservieren
  2. Infect - Ressourcen nutzen
  3. Retreat - Ressourcen freigeben
- OctoPOS und iRTSS bieten Software-Grundlage
- C-Schnittstelle
- Unterstützung der Programmiersprachen C, C++ und X10

Rust - Motivation

- Sichere Speicherzugriffe ohne Garbage Collector
- Vermeidung undefinierten Verhaltens
- Effiziente und weniger fehleranfällige Parallelberechnung
- Höhere Abstraktionen, um den Einstieg zu erleichtern
- Speichersicherheit und Abstraktionen sollen nicht auf Kosten der Leistung erreicht werden

Rust - Ownership, Move-Semantik und Referenzen

- Ownership
  - Das zentrale Alleinstellungsmerkmal der Programmiersprache
  - Jeder Speicherbereich wird nur einer einzigen Variable zur Verfügung gestellt
  - Beim Verlassen des Geltungsbereichs wird der Speicherbereich freigegeben
  - => Affine Typen

Rust - Ownership, Move-Semantik und Referenzen

- Ownership
  - Das zentrale Alleinstellungsmerkmal der Programmiersprache
  - Jeder Speicherbereich wird nur einer einzigen Variable zur Verfügung gestellt
  - Beim Verlassen des Geltungsbereichs wird der Speicherbereich freigegeben
  - => Affine Typen
- Move-Semantik
  - Ownership kann auf andere Variablen übertragen werden
  - Ursprüngliche Variable ist nach einem „Move“ nicht mehr verwendbar

Rust - Ownership, Move-Semantik und Referenzen

- Ownership
  - Das zentrale Alleinstellungsmerkmal der Programmiersprache
  - Jeder Speicherbereich wird nur einer einzigen Variable zur Verfügung gestellt
  - Beim Verlassen des Geltungsbereichs wird der Speicherbereich freigegeben
  - => Affine Typen
- Move-Semantik
  - Ownership kann auf andere Variablen übertragen werden
  - Ursprüngliche Variable ist nach einem „Move“ nicht mehr verwendbar
- Referenzen
  - Unendliche unveränderliche Referenzen
  - Nur eine veränderliche Referenz
  - Move einer Variable nicht möglich wenn Referenzen existieren

Ownership und Move-Semantik

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");




...
```

Ownership und Move-Semantik

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;



...
```

Ownership und Move-Semantik

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;
  f(a);


...
```

```
error[E0382]:
use of moved value: ‘a‘
```

Ownership und Move-Semantik

```
fn f(s: String) { ... }
...
  let a = String::from("Hello␣World!");
  let b = a;
  f(b);


...
```

Ownership und Move-Semantik

```
fn f(s: String) { ... }
...
  let a = String::from("Hello World!");
  let b = a;
  f(b);
  f(b);

...
```

```
error[E0382]:
use of moved value: ‘b‘
```

Referenzen

```
fn g(s: &mut String) { ... }
...
  let mut x = String::from("Hello␣World!");
...
```

Referenzen

```
fn g(s: &mut String) { ... }
...
  let mut x = String::from("Hello␣World!");
  let x_ref1 = &x;




...
```

Referenzen

```
fn g(s: &mut String) { ... }
...
  let mut x = String::from("Hello␣World!");
  let x_ref1 = &x;
  let mut y = x;




...
```

```
error[E0505]:
cannot move out of ‘x‘ because it is borrowed
```

Referenzen

```
fn g(s: &mut String) { ... }
...
  let mut x = String::from("Hello␣World!");
  let x_ref1 = &x;
  let x_ref2 = &x;




...
```

Referenzen

```
fn g(s: &mut String) { ... }
...
  let mut x = String::from("Hello␣World!");
  let x_ref1 = &x;
  let x_ref2 = &x;
  let x_ref3 = &mut x;



...
```

Referenzen

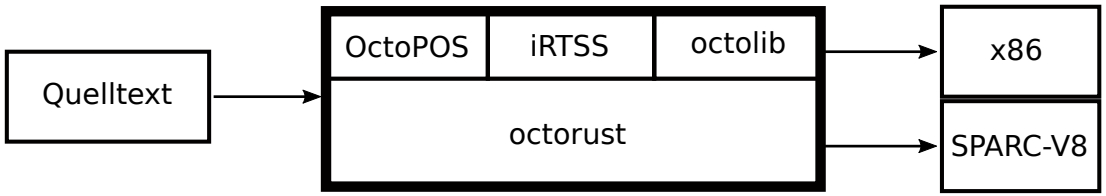
```
fn g(s: &mut String) { ... }
...
  let mut x = String::from("Hello␣World!");
  let x_ref1 = &x;
  let x_ref2 = &x;
  let x_ref3 = &mut x;
  let x_ref4 = &mut x;


...
```

```
error[E0499]:
cannot borrow ‘x‘ as mutable more than once at a time
```

octorust

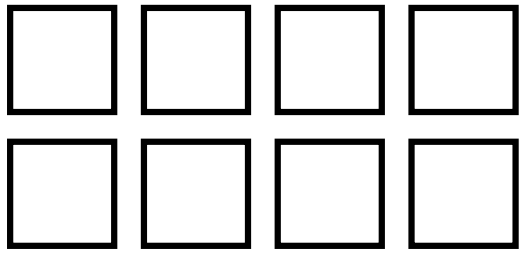
- Hilfsprogramm zum Kompilieren von invasiven Rust-Programmen
- Unterstützt die x86 und SPARC-V8 Architekturen
- Automatische Verlinkung mit iRTSS/OctoPOS
- Automatisches Einbinden von octolib
- ca. 650 Zeilen Code (Python)



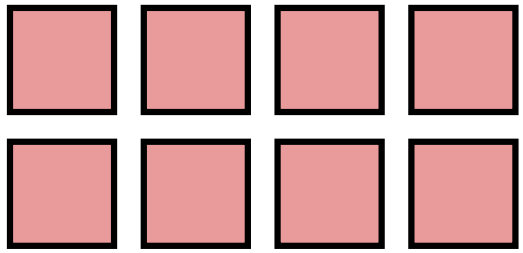
octolib

- Rust-Bibliothek mit invasiven Strukturen und Funktionen
- Direkte C-Rust Bindings
- Rust-spezifische Anpassungen
  - Objektorientierte Constraints
  - Verwendung von Closures
  - AgentClaim-Struktur
- ca. 750 Zeilen Code (Rust)

Beispiel eines invasiven Rust-Programms

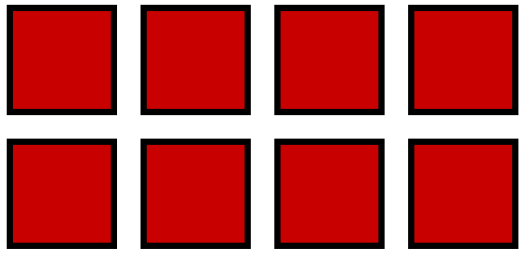


Beispiel eines invasiven Rust-Programms



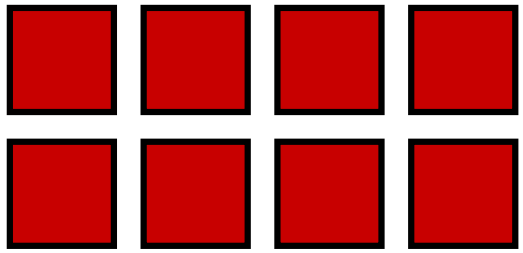
```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);
```

Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);
```

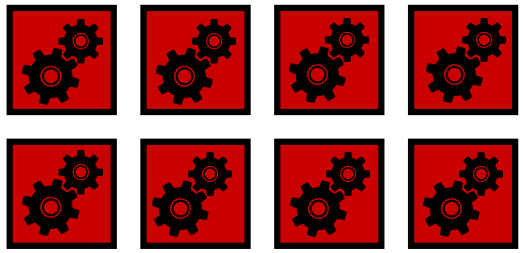
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
```

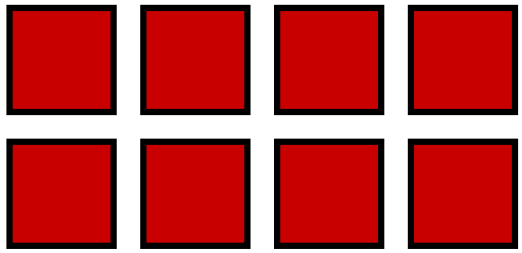
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
```

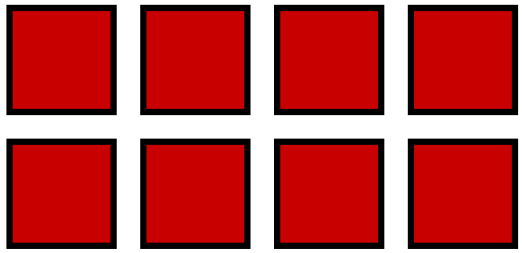
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
```

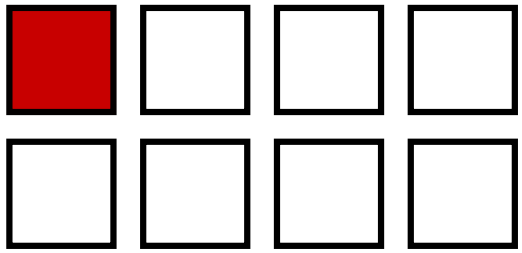
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
```

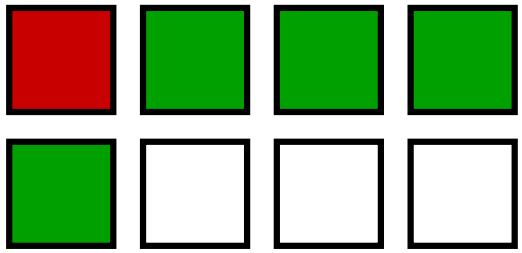
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
```

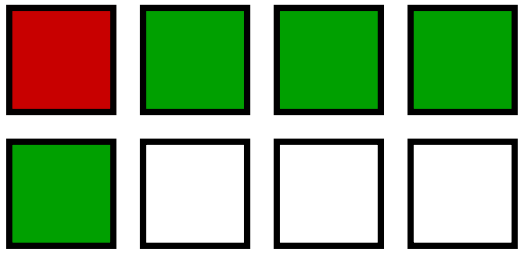
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
```

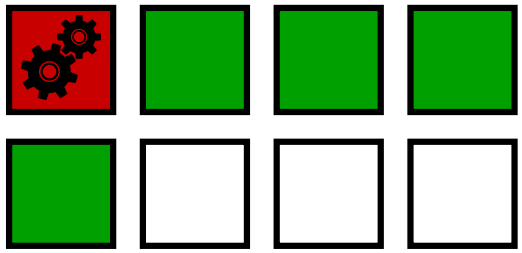
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
```

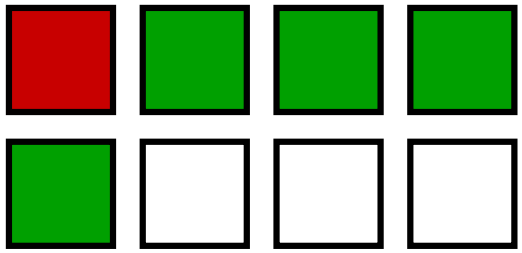
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
```

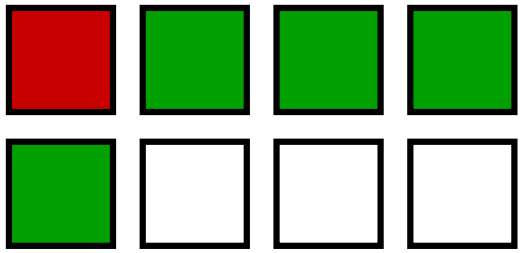
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
```

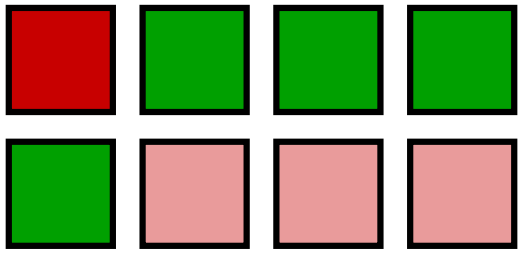
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
claim.reinvade(Constraints::new(4, 8));
```

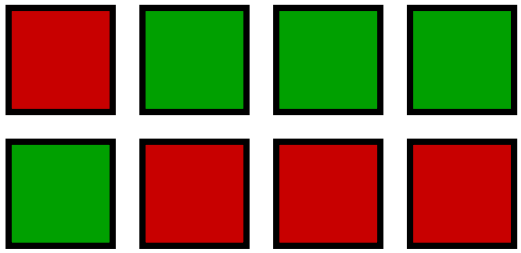
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
claim.reinvade(Constraints::new(4, 8));
```

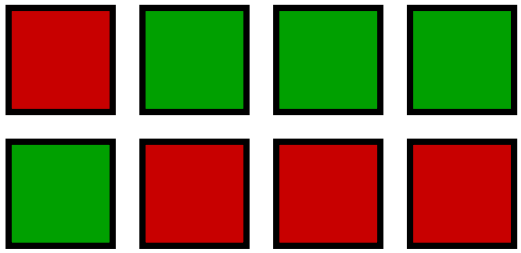
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
claim.reinvade(Constraints::new(4, 8));
```

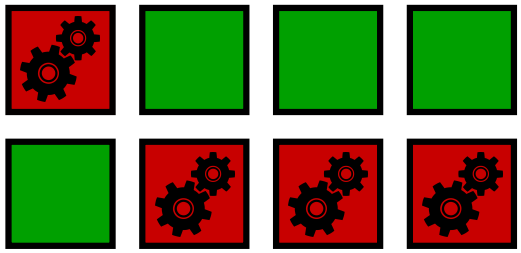
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
claim.reinvade(Constraints::new(4, 8));
claim.infect(ilet_fn);
```

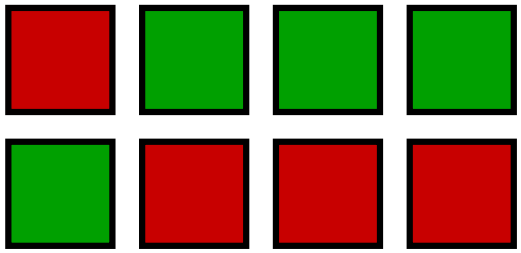
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
claim.reinvade(Constraints::new(4, 8));
claim.infect(ilet_fn);
```

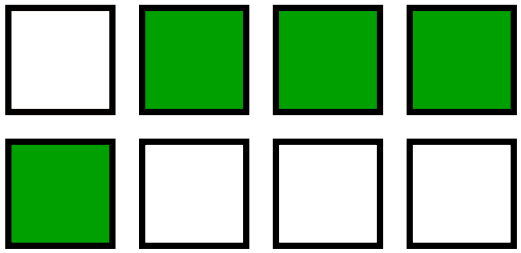
Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
claim.reinvade(Constraints::new(4, 8));
claim.infect(ilet_fn);
// Implizites Retreat beim Verlassen des Geltungsbereichs
```

Beispiel eines invasiven Rust-Programms



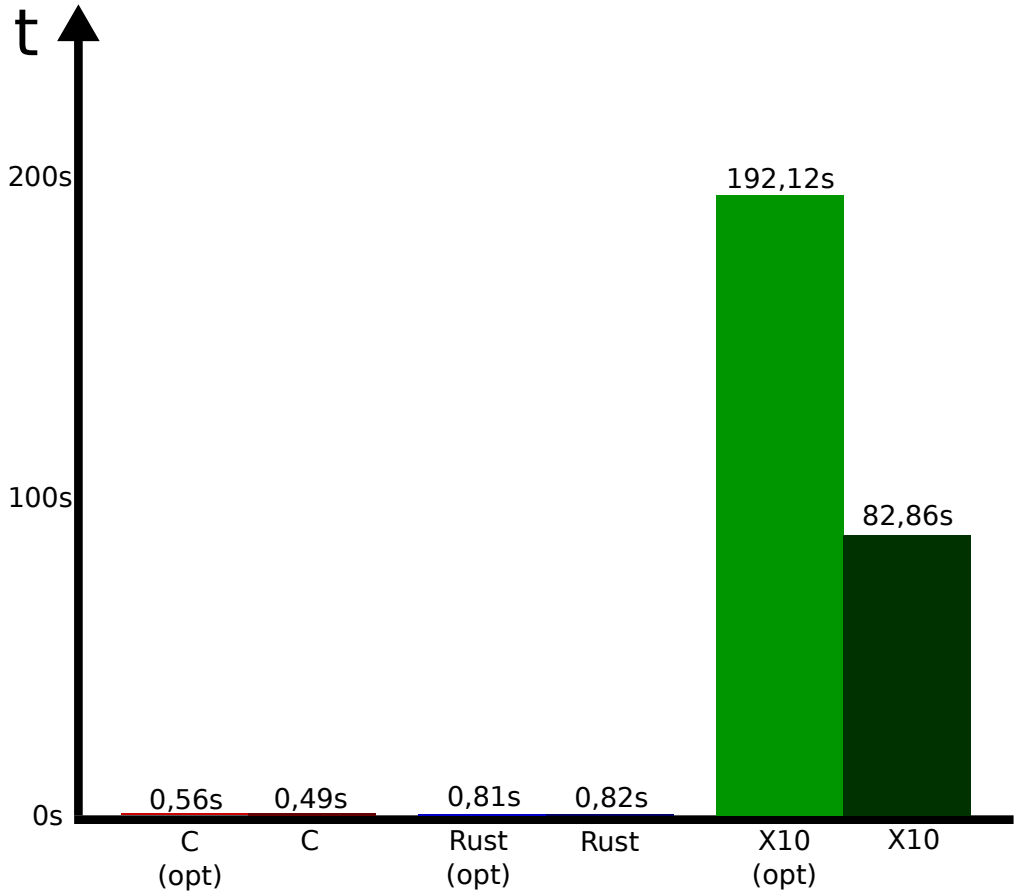
```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
claim.reinvade(Constraints::new(4, 8));
claim.infect(ilet_fn);
// Implizites Retreat beim Verlassen des Geltungsbereichs
```

Evaluation

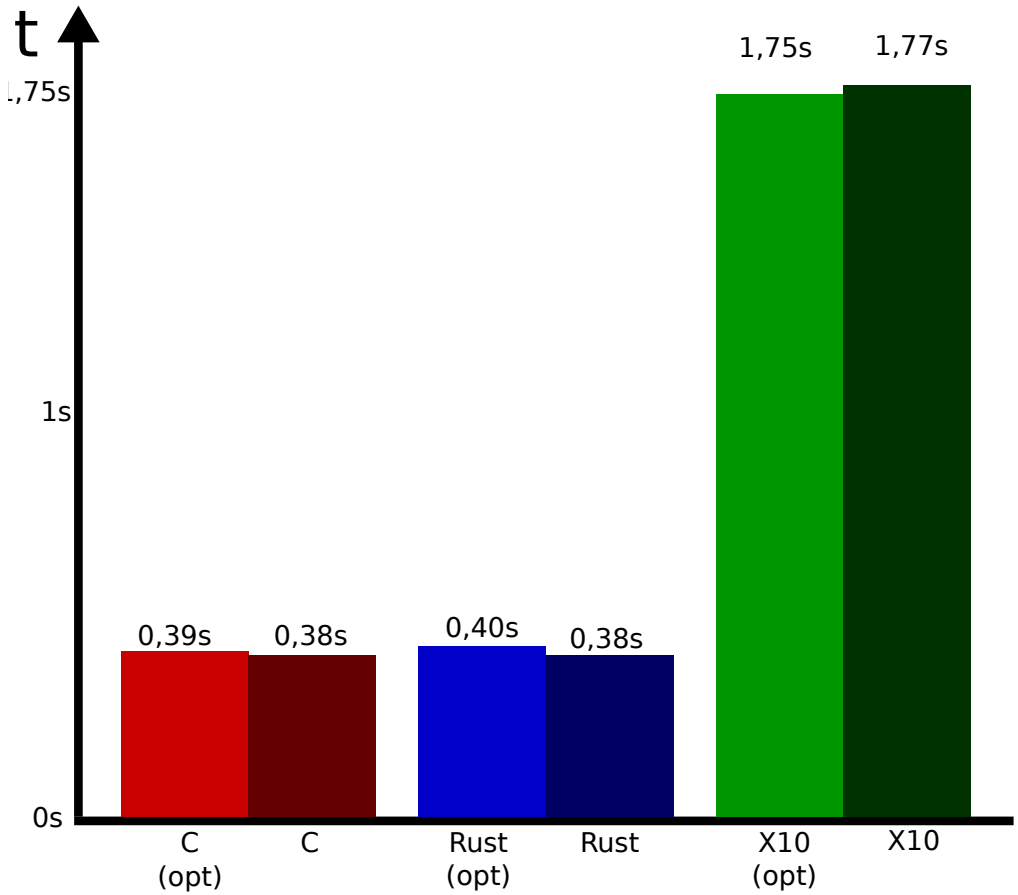
- Rust wird mit C und X10 verglichen
- Wiederholte Laufzeitmessungen
- Betrachtung von Programmen mit und ohne Compiler-Optimierungen
- Unterschiedliche Programme
  - Kompilierungsdauer
  - Anlaufzeit
  - Parallele Primzahlenberechnung
  - Allokation von Objekten

Kompilierungsdauer



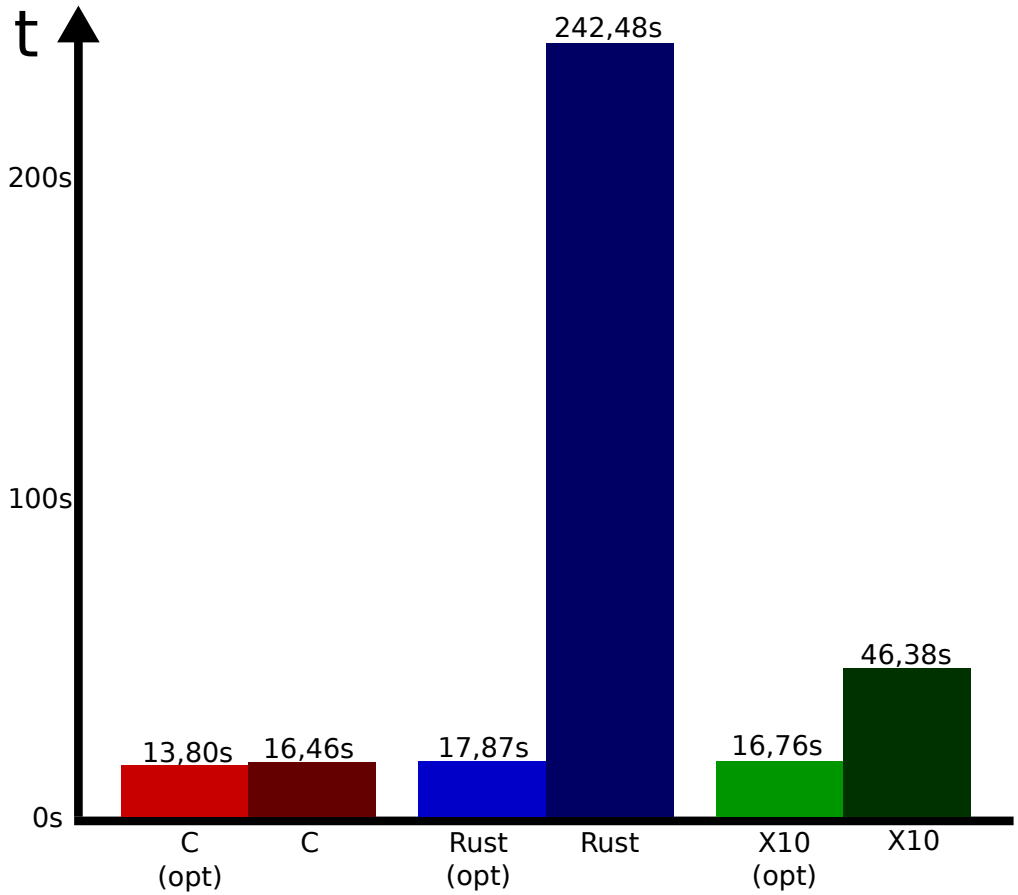
- Zeitersparnis beim Kompilieren von Rust-Programmen im Vergleich mit X10-Programmen

Anlaufzeit



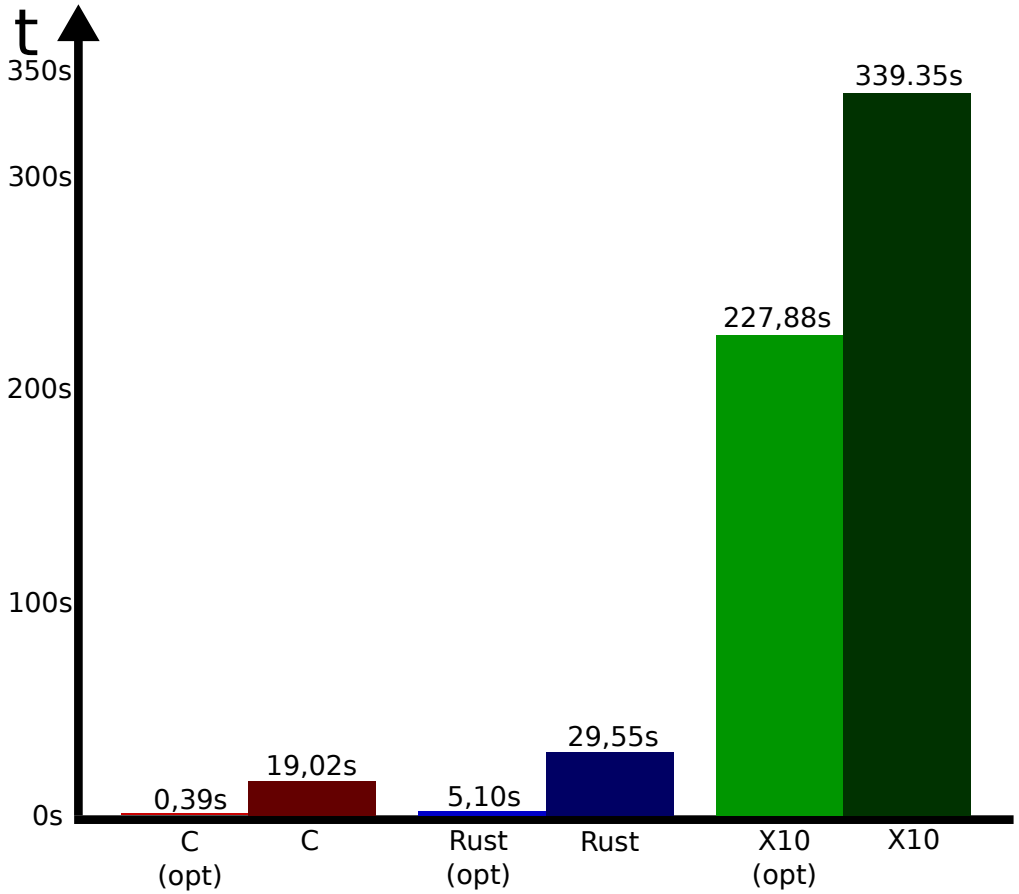
- Rust und C benötigen ca. 1.4 Sekunden weniger um zu starten

Parallele Primzahlenberechnung



- Rust weist bei Berechnung von Primzahlen keine Leistungsvorteile auf

Allokation von Objekten



- C und Rust weisen eine mindestens 10-fach bessere Laufzeit als X10 auf

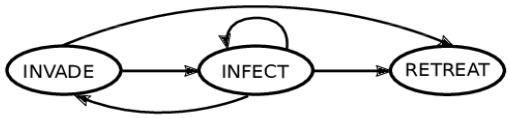
Zusammenfassung und Fazit

- Invasives Rechnen ermöglicht ressourcenbewusstes Parallelrechnen
- Rust bietet Speichersicherheit ohne Garbage Collector
- Rust schützt vor undefinertem Verhalten
- octorust und octolib ermöglichen den Einsatz von Rust im invasiven Rechnen
- Rust weist in speicherintensiven Szenarien bessere Laufzeiten als X10 auf

Vielen Dank für ihre Aufmerksamkeit

Karlsruher Institut für Technologie

Invasives Rechnen

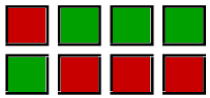


- Ressourcenbewusstes Programmieren
- 3 Phasen:
  1. Invade - Ressourcen reservieren
  2. Infect - Ressourcen nutzen
  3. Retreat - Ressourcen freigeben
- OctoPOS und iRTSS bieten Software-Grundlage
- C-Schnittstelle
- Unterstützung der Programmiersprachen C, C++ und X10

Rust - Ownership, Move-Semantik und Referenzen

- Ownership
  - Das zentrale Alleinstellungsmerkmal der Programmiersprache
  - Jeder Speicherbereich wird nur einer einzigen Variable zur Verfügung gestellt
  - Beim Verlassen des Geltungsbereichs wird der Speicherbereich freigegeben
  - => Affine Typen
- Move-Semantik
  - Ownership kann auf andere Variablen übertragen werden
  - Ursprüngliche Variable ist nach einem „Move“ nicht mehr verwendbar
- Referenzen
  - Unendliche unveränderliche Referenzen
  - Nur eine veränderliche Referenz
  - Move einer Variable nicht möglich wenn Referenzen existieren

Beispiel eines invasiven Rust-Programms



```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

let ilet_fn = |param: *mut c_void| {
  ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
claim.reinvade(Constraints::new(4, 8));
claim.infect(ilet_fn);
// Implizites Retreat beim Verlassen des Geltungsbereichs
```


Zusammenfassung und Fazit

- Mithilfe des invasiven Rechnens kann ressourcenbewusst programmiert werden
- Rust bietet Speichersicherheit ohne Garbage Collector
- Rust weist in speicherintensiven Szenarien bessere Laufzeiten als X10 auf
- Im Gegensatz zu C verhindert Rust undefiniertes Verhalten