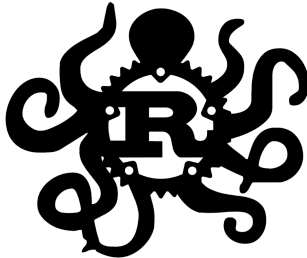
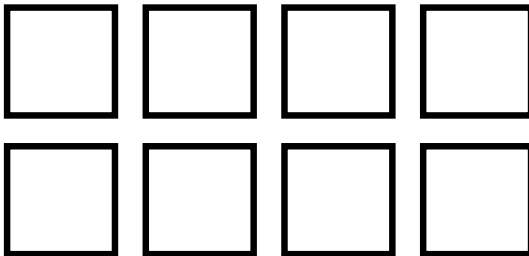


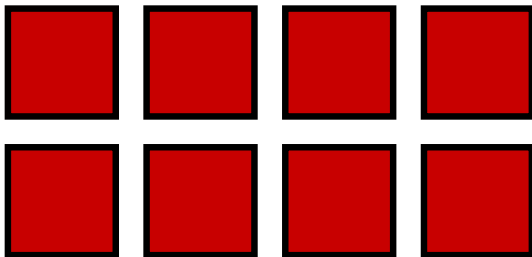
# Invasives Rust

Hermann Heinz Erich Krumrey

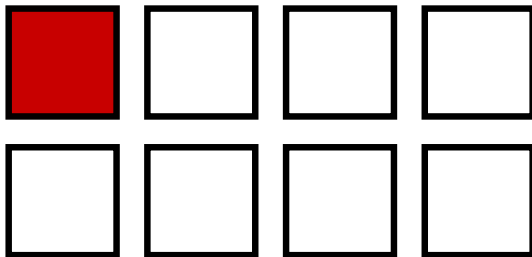
Lehrstuhl Programmierparadigmen, IPD Snelting



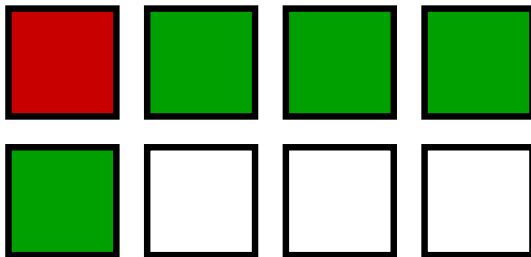




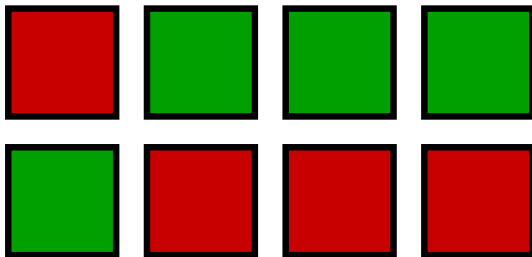
1. **Programm 1** beginnt Ausführung auf 8 Recheneinheiten



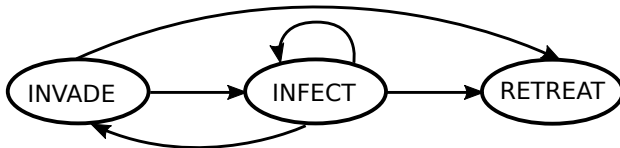
1. **Programm 1** beginnt Ausführung auf 8 Recheneinheiten
2. **Programm 1** sendet Ergebnisse über das Netzwerk



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



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



- 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

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



## ■ 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

## ■ 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

## ■ 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

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

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

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

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

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

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

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



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

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

```
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
```

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

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

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

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

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

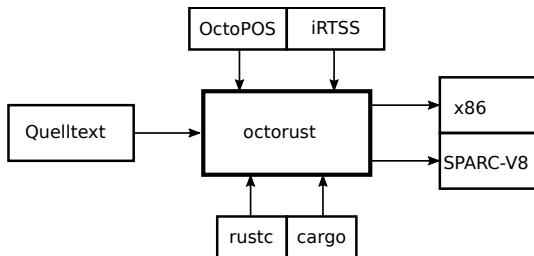
- Ähneln Interfaces aus anderen Sprachen
- Können Verhalten bezüglich Ownership und Move-Semantik beeinflussen



- Ähneln Interfaces aus anderen Sprachen
- Können Verhalten bezüglich Ownership und Move-Semantik beeinflussen
- Drop
  - Destruktor
  - `drop()`-Methode wird beim Verlassen des Geltungsbereichs aufgerufen
  - Verwendung zum Beispiel zur Schließung einer Netzwerkverbindung

- Ähneln Interfaces aus anderen Sprachen
- Können Verhalten bezüglich Ownership und Move-Semantik beeinflussen
- Drop
  - Destruktor
  - `drop()`-Methode wird beim Verlassen des Geltungsbereichs aufgerufen
  - Verwendung zum Beispiel zur Schließung einer Netzwerkverbindung
- Copy
  - Erlaubt Copy-Semantik statt Move-Semantik
  - Jedes mal, wenn ein Move geschehen würde, wird eine bitweise Kopie ausgeführt
  - Primitive Datentypen wie `i32` implementieren dieses Trait

- 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)



- 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)

# Rust Bindings zur OctoPOS/iRTSS C-Schnittstelle

```
uint32_t get_value(void *data)
```

```
extern crate libc;  
use libc::c_void;
```

# Rust Bindings zur OctoPOS/iRTSS C-Schnittstelle

```
uint32_t get_value(void *data)
```

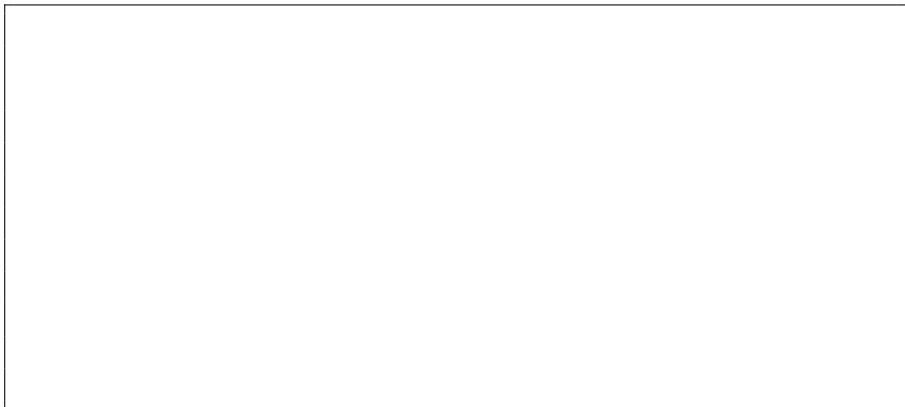
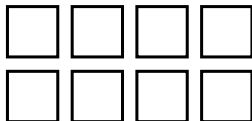
```
extern crate libc;  
use libc::c_void;  
  
extern "C" {  
  
}
```

# Rust Bindings zur OctoPOS/iRTSS C-Schnittstelle

```
uint32_t get_value(void *data)
```

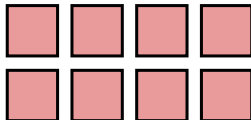
```
extern crate libc;  
use libc::c_void;  
  
extern "C" {  
    fn get_value(data: *mut c_void) -> u32;  
}
```

# Beispiel eines invasiven Rust-Programms



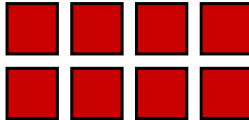


# Beispiel eines invasiven Rust-Programms



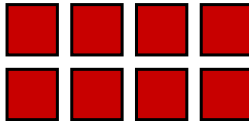
```
let constraints = Constraints::new(4, 8);  
let claim = AgentClaim::new(constraints);
```

# Beispiel eines invasiven Rust-Programms



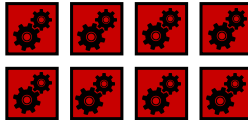
```
let constraints = Constraints::new(4, 8);  
let claim = AgentClaim::new(constraints);
```

# Beispiel eines invasiven Rust-Programms



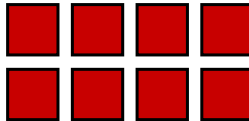
```
let constraints = Constraints::new(4, 8);  
let claim = AgentClaim::new(constraints);  
  
let ilet_fn = |param: *mut c_void| {  
    ...  
}  
claim.infect(ilet_fn);
```

# Beispiel eines invasiven Rust-Programms



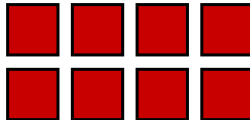
```
let constraints = Constraints::new(4, 8);  
let claim = AgentClaim::new(constraints);  
  
let ilet_fn = |param: *mut c_void| {  
    ...  
}  
claim.infect(ilet_fn);
```

# Beispiel eines invasiven Rust-Programms



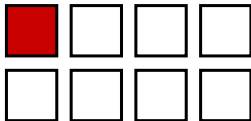
```
let constraints = Constraints::new(4, 8);  
let claim = AgentClaim::new(constraints);  
  
let ilet_fn = |param: *mut c_void| {  
    ...  
}  
claim.infect(ilet_fn);
```

# 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));
```

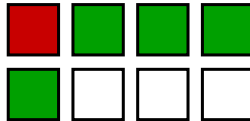
# Beispiel eines invasiven Rust-Programms



```
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

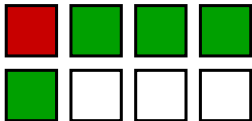


```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

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

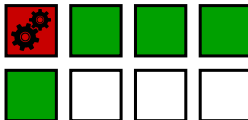


# Beispiel eines invasiven Rust-Programms



```
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);
```

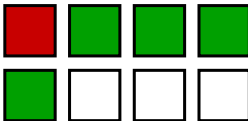
# Beispiel eines invasiven Rust-Programms



```
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);
```

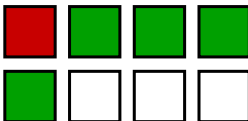
# Beispiel eines invasiven Rust-Programms



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

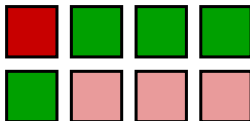
let ilet_fn = |param: *mut c_void| {
    ...
}
claim.infect(ilet_fn);
claim.reinvade(Constraints::new(1, 1));
claim.infect(network_fn);
```

# Beispiel eines invasiven Rust-Programms



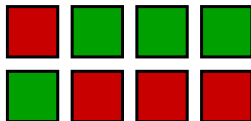
```
let constraints = Constraints::new(4, 8);  
let claim = AgentClaim::new(constraints);  
  
let ilet_fn = |param: *mut c_void| {  
    ...  
}  
claim.infect(ilet_fn);  
claim.reinvade(Constraints::new(1, 1));  
claim.infect(network_fn);  
claim.reinvade(Constraints::new(4, 8));
```

# Beispiel eines invasiven Rust-Programms



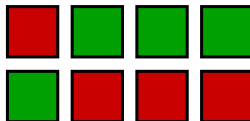
```
let constraints = Constraints::new(4, 8);  
let claim = AgentClaim::new(constraints);  
  
let ilet_fn = |param: *mut c_void| {  
    ...  
}  
claim.infect(ilet_fn);  
claim.reinvade(Constraints::new(1, 1));  
claim.infect(network_fn);  
claim.reinvade(Constraints::new(4, 8));
```

# 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));
```

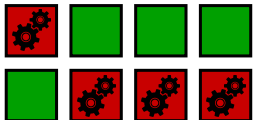
# Beispiel eines invasiven Rust-Programms



```
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

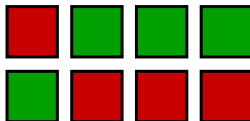


```
let constraints = Constraints::new(4, 8);
let claim = AgentClaim::new(constraints);

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);
```



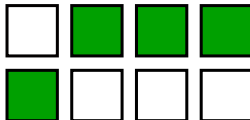
# Beispiel eines invasiven Rust-Programms



```
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@
```

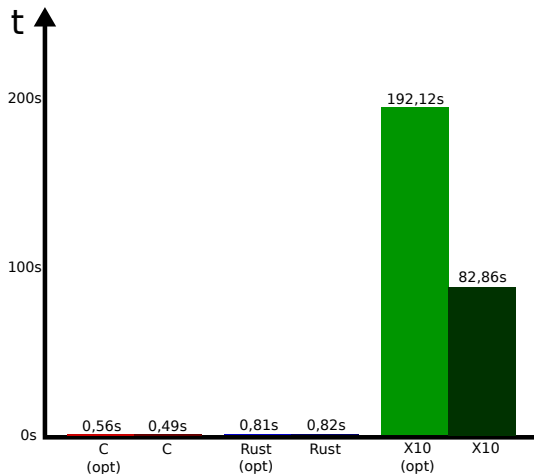
# Beispiel eines invasiven Rust-Programms



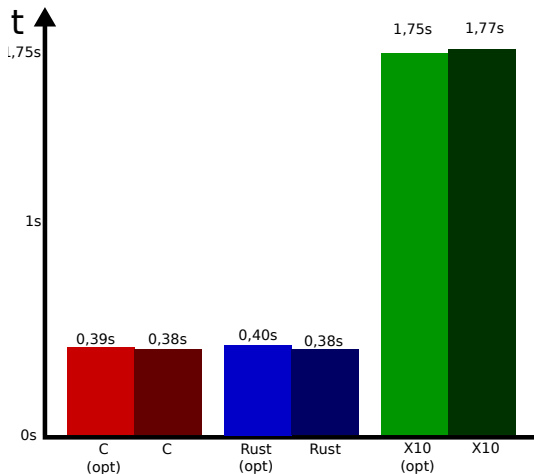
```
let constraints = Constraints::new(4, 8);  
let claim = AgentClaim::new(constraints);  
  
let ilet_fn = |param: *mut c_void| {  
    ...  
}  
claim.infect(ilet_fn);  
claim.reinvade(Constraints::new(1, 1));  
claim.infect(network_fn);  
claim.reinvade(Constraints::new(4, 8));  
claim.infect(ilet_fn);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

*Geltungsbereichs@*

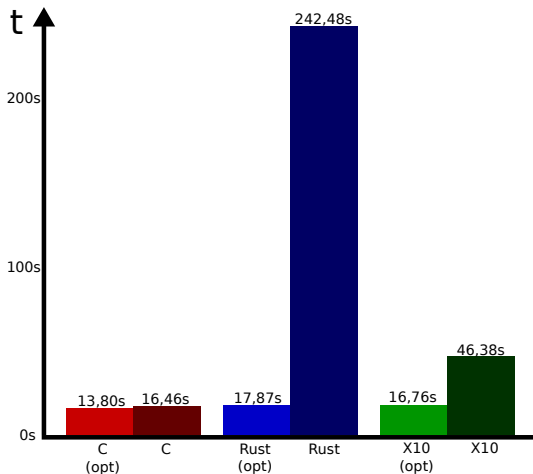
- Wiederholte Laufzeitmessungen
- Betrachtung von Programmen mit und ohne Compiler-Optimierungen
- Unterschiedliche Programme
  - Kompilierungsdauer
  - Anlaufzeit
  - Parallele Primzahlenberechnung
  - Allokation von Objekten



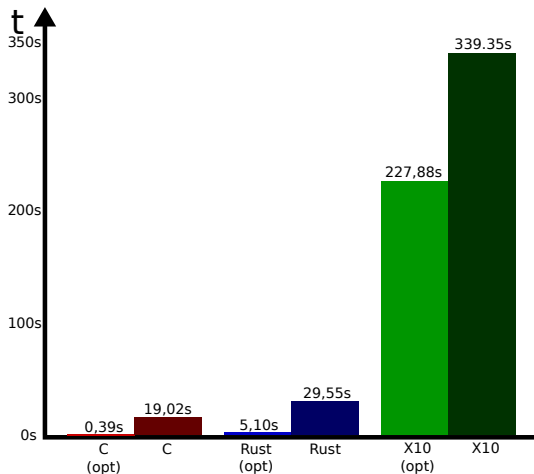
- Zeitersparnis beim Kompilieren von Rust-Programmen im Vergleich mit X10-Programmen



■ Rust und C benötigen ca. 1.5 Sekunden weniger um zu starten



■ Rust weist bei Berechnung von Primzahlen keine Leistungsvorteile auf



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

- 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



# Vielen Dank für ihre Aufmerksamkeit!

## 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

12 26.10.2013 Hermann Heinz Erich Krumrey - Invasives Rust

IPD

## 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

5 26.10.2013 Hermann Heinz Erich Krumrey - Invasives Rust

IPD

## 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
```

12 26.10.2013 Hermann Heinz Erich Krumrey - Invasives Rust

IPD

## 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

18 26.10.2013 Hermann Heinz Erich Krumrey - Invasives Rust

IPD