Rusts typ- och minnessäkerhet

Alexander Westerlund

40220

Åbo Akademi

Fakulteten för naturvetenskaper och teknik

Handledare: Ivan Porres

03-04-2019

Innehåll

Referat

Minnes- och typsäkerhet säkerställer att en stor kategori buggar inte uppstår. I denna avhandling undersöks hur Rust förverkligar minnessäkerhet samt hur detta används i praktiken. Avhandlingens sekundära mål är att undersöka genomförbarheten av att ersätta osäkra systemprogrammeringsspråk med Rust samt om Rusts minnessäkringsåtgärder fungerar i verkligheten.

Sökord: rust, minnessäkerhet

1. Inledning

Dagens mjukvarusystem blir alltmer invecklade och svåra att upprätthålla. För att minimera mängden buggar i dessa mjukvarusystem har man skapat språk som är minnessäkra. Säkerhet i denna kontext betyder avsaknad av odefinierat beteende. Minnessäkerhet innebär att man är skyddad från buggar som kan ske vid hantering av minnesåtkomst [3].

Typsäkerhet betyder att man säkerställer att t.ex. en variabel eller ett objekt är datatypen den borde vara. Typsäkerhet medför därför också att programmet är bättre definierat i sin helhet, vilket alltid eftersträvas i all mjukvara.

I programmering har man inte alltid möjligheten att använda tyngre programmeringsekosystem som säkerställer minnessäkerhet genom ett sophämtningssystem. Orsaken till detta kan vara t.ex. hårdvarubegränsningar såsom begränsat arbetsminne och begränsad processorkraft.

Språk som anses vara icke-minnessäkra är till exempel C och C++. De flesta icke-minnessäkra språken kan dock göras säkra med tillräckligt arbete. Exempel på språk som är utvecklade att vara minnessäkra språk är Rust, Cyclone och Java. Cyclone är språket som liknar mest på Rust, men övergavs år 2006 när version 1.0 släpptes. Det man försökte åstadkomma med Cyclone var en dialekt av C som skulle erbjuda en lösning som tillåter lätt ersättning av osäker C-kod med säker Cyclone-kod utan att vara tvungen att programmera om hela mjukvaran i ett nytt språk. Tack vare att Cyclone inte var ett nytt språk, utan endast en dialekt av C, hade man vissa begränsningar i utvecklingen av projektet vilket resulterade i att språkets semantik inte var optimal. Detta är en av orsakerna varför Rusts utvecklingsteam valde att skapa ett nytt språk istället för en dialekt eller ett bibliotek till ett existerande språk. [4]

I den här avhandlingen förklaras och undersöks metoderna Rust använder för att säkerställa minnes- och typsäkerhet. När det är lämpligt kommer Rust jämföras med C, C++, Java samt mer sällan Cyclone för att illustrera vad som gör Rust unikt bland ett flertal språk som löst, och inte löst problemet med inbyggs minnessäkerhet. I kapitlet ”Rusts lösningar” visas hur Rust uppnått minnes- och typsäkerhet. [2, 3, 4].

2. Exempel på minnessäkerhetsrelaterade buggar

Det finns många buggar som kan uppstå på grund av bristfällig minnessäkerhet. I detta kapitel förklaras ett fåtal av dessa buggar. Alla dessa buggar resulterar i odefinierat beteende. [1, 4]

2.1. Buffertöverfyllning

Buffertöverfyllning innebär att man går över en bufferts gräns och därav skriver data till minnesplatsen som kommer efter bufferten. Buffertöverfyllningar kan bland annat orsaka korrupt data, programkrashar eller exekvering av skadlig kod. [1, 4].

2.2. Användning efter frigöring

Användning efter frigöring betyder att man använder en minnespekare (minnesreferens/variabel) efter att dess minnesplats har frigjorts [4]. Detta medföljer odefinierat beteende eftersom vad som helst kan finnas vid minnesplatsen vid avreferering av pekaren. I det bästa fallet returneras minnesadressen till operativsystemet och programmet kraschar tack vare segmenteringsfel. Om programmet inte kraschar kan programmet hämta vilken data som helst som allokerats inom processen.

Ett vanligt sätt för denna bugg att uppstå är genom så kallade dinglande pekare. En dinglande pekar mot en inkorrekt minnesadress. Dinglande pekare uppstår till exempel när man förstör ett objekt utan att ta bort alla existerande pekare som pekar mot objektets minnesadress. Detta resulterar i onödig användning av minne och potentiell exekvering av skadlig kod. [5]

De flesta språk idag löser användning efter frigöring genom att ha ett sophämtningssystem. Ett exempel på användning efter frigöring, illustrerad i C, finns nedan i **Error! Reference source not found.**. [4]

```
int *var1 = malloc(sizeof(int)); // Allokerar minne

free(var1); // Frigör minnet bakom pekaren

printf("%d", *var1); // Vi vet inte vad som finns bakom pekaren längre
```

Figur 1 - Användning efter frigöring

2.3. Avreferering av nullpekare

En nullpekare pekar mot en minnesadress som inte existerar. Avreferering av en sådan pekare resulterar i språk såsom Java, C och C++ med odefinierat beteende. Vanligen kraschar program när man försöker avreferera en nullpekare. Det må finnas flera sätt att motverka avreferering av nullpekare, men det lättaste sättet är att inte tillåta skapelsen av nullpekare till att börja med; det är just detta sätt som Rust använder sig av [6]. Nedan i **Error! Reference source not found.** finns ett exempel på avreferering av nullpekare i C.

```
int *pointer = NULL;

// pekaren "pointer" avrefereras och programmet kraschar
printf("Pointer: %x", pointer);
```

Figur 2 - Avreferering av nullpekare

2.4. Data races

Ett data race är en bugg där flera trådar försöker få åtkomst till samma data, dvs ifall tråd 2 är beroende av att tråd 1 hämtar data före tråd 2 kan odefinierat beteende uppstå genom att fel tråd hämtar data först. Många optimeringar som normalt kan göras av kompilatorn är inte användbara om man behandlar kod som arbetar med multitrådning. Det är väldigt svårt att lösa data race problemet. [4]

C och C++ har inget automatiserat sätt att motverka data races, utan programmeraren måste implementera detta själva. Dessa två språk tillåter programmeraren lösa data race problemet genom att använda atomiska (Eng: Atomic) operationer som har nackdelen att de hämmar potentiell optimering i kompilatorn. Det finns ingenting i dessa språk som motverkar felanvändning eller bortlämnande av dessa atomiska operationer och därför finns ingen garanti i själva programmeringsspråket att man inte har ett data race gömt i koden. [3, 4, 5]

Vissa programmeringsspråk löser problemet med data races genom att förbjuda delning mellan trådar, föränderlighet eller samtidighet. Till exempel Java tillåter alla tre av dessa, samt garanterar att det inte finns data races genom att ha starka atomicitetsgarantier.

3. Introduktion till Rust

För att ha komplett förståelse för senare kapitel i avhandlingen krävs en viss grundkunskap över Rusts uppbyggnad. Rusts grundläggande egenskaper och funktioner drar till stor del inspiration från andra språk och kan därför se bekanta ut, till exempel använder Rust samma grundläggande syntax för for- och while slingor samt villkorliga hoppinstruktioner (nyckelorden *if* och *else*). [3, 4]

Rust är ett programmeringsspråk som fokuserar på säkerhet, hastighet och samtidighet. Dessa mål uppnås utan användningen av en sophämtare. Avsaknaden av en sophämtare resulterar i att Rust är lätt att använda tillsammans med andra språk och existerande mjukvara. Rust är ett lågnivåspråk och lämpar sig därför för systemprogrammering, dock inte endast systemprogrammering. [3, 4, 5, 6]

Rust har en egen pakethanterare som heter Cargo. Cargo tillåter mjukvaruutvecklare att enkelt skapa nya paket(bibliotek). Paket i Rustvärlden kallas för ”crates”(Spjällåda eller packkorg på svenska). Det är väldigt svårt att utveckla ett större projekt i Rust utan användningen av externa crates. Rust stöder också namnrymder, namnrymder används också flitigt hos Rusts inbyggda bibliotek. [3]

3.1. Variabler

Variabler i Rust deklareras med kommandot ”let” (sve. låt; t.ex. låt x vara 5). Per standard är alla variabler oföränderliga. För att deklarera en föränderlig variabel använder man nyckelordet ”mut” som står för ”mutable” (sve. föränderlig). Efter variabelnamnet kan man specificera man datatypen. Specificerar man inte datatypen kommer kompilatorn automatiskt välja den mest troliga datatypen för värdet. Exempel på variabeldeklaration finns i figur 3 nedan. I figuren deklarerar vi en mängd olika variabler, både med explicit- och implicit specificerade datatyper.

```
let a = "Hej!";         // Oföränderlig variabel "a" med automatisk datatyp "str".
let b: str = "Hej 2!";  // Oföränderlig variabel "b" med explicit datatyp "str".
let mut c = 1;          // Föränderlig variabel "c" med automatisk datatyp, blir troligen  typ "i32".
let mut d: i32 = 3;     // Föränderlig variabel "d" med explicit datatyp "i32".
```

Figur 3- Variabeldeklaration

3.2. Slingor

Det finns tre olika sorts slingor i Rust [3]:

1. Ändlös slinga (Nyckelordet ”loop”) som måste avbrytas med nyckelordet *break*.
2. Upprepning med villkor (Nyckelordet ”while”)
3. For-slinga (Nyckelordet ”for”)

Slingor i Rust fungerar likadant som i C med ett par undantag. I C existerar inte ändlösa slingor som ett reserverat nyckelord för det ändamålet, utan man har varit tvungen att skriva ”while (true)”. I Rust kan man använda for-slingor som ”for each”-slingor, dock är det endast möjligt att göra detta på datatypen *slice*. Man kan till exempel omvandla en uppställning av värden till en slice genom att använda funktionen *iter()*. Ett par exempel på användningen av for-loopar samt metoden *iter()* finns nedan i figur 4.

```
let mut loopforever = true;
let mut nr = 27;
let array = ["a", "b", "c"];

// Evig slinga som manuellt måste avbrytas
loop {
    if !loopforever {
        break;
    }

    println!("Loopyloop!");
    loopforever = false;
}

// Vanlig while-slinga
while nr < 30 {
    println!("While loop! {}", nr);
    nr = nr + 1;
}

// Vanlig for-slinga från 13 till 37

for nummer in 13..37 { println!("Nummer i for-slinga: {}", nummer) }
// "For each"-slinga
for letter in array.iter() { println!("Bokstav: {}", letter) }
```

Figur 4 - Slingor

3.3. Funktioner

Funktioner i Rust deklareras som i de flesta imperativa språk, man använder nyckelordet ”fn” för att markera början av en funktionsdeklaration. Till exempel nedan i figur 5 deklarerar vi en funktion vars namn är ”do_something” som tar in 2 argument som är värden är av typerna *i32* och *String* samt returnerar en *String* ur funktionen.

```
let a = 3;
let b = String::from("Hej");

fn do_something(a: i32, b: String) -> (String) {
    println!("Argument 1: {}, Argument 2: {}", a, b);

    String::from("Returnerad sträng ur funktionen!")
}

println!("Ur funktionen: {}", do_something(a, b));
```

Figur 5 - Funktionsdeklaration

Rust har stöd för generiska typer i funktionsdeklarationer. Generiska typer som definierats i funktionsdeklarationen ses som generiska genom funktionen fastän det finns en konkret typ(struktur) som heter samma sak. Ett exempel på en funktion med två generiska datatyper finns nedan i figur 6. [3]

```
fn do_something<T, O>(x: T, y: O) { ... }
```

Figur 6 - Generiska typer

3.4. Egenskaper och Strukturer

En egenskap (eng. Trait) i Rust är en samling metoder som definieras för en viss struktur. Strukturen (eng. Struct) som implementerar en viss egenskap får åtkomst att kalla på alla metoder som definierats för just den egenskapen. Till exempel har man en datatyp *Car* som implementerar en egenskap *Motor*, får instanser av *Car* förmågan att kalla på alla funktioner som definieras av *Motor*. Detta är väldigt likt gränssnitt(interfaces) i Java. [3]

Strukturers huvudsakliga funktion i Rust grupperar en mängd variabler under ett gemensamt namn, exakt som strukturer i C. Man kan också kalla strukturer för datatyper

i Rust, orden är helt utbytbara med varandra. Ett exempel på definiering, implementering samt användning av datatyper och egenskaper finns nedan i figur 7. [3]

Rust har stöd för enumerationer genom nyckelordet *enum*. Enumerationer är en datatyp som kan vara av flera olika varianter. Enumerationer används flitigt av Rusts standardbibliotek inom felhantering. Felhantering i Rust kommer diskuteras i ett skiljt kapitel senare i avhandlingen. För att exekvera olik kod beroende på variant använder man nyckelordet *match*. Match exekverar kod beroende på den egentliga varianten av en enumeration. nedan i figur 8 definieras en enumeration som heter VehicleEvent. VehicleEvent har tre olika varianter, *Accelerate*, *Brake* och *ShoutAt*. I figuren definieras också en funktion *example_match* som tar in ett VehicleEvent som enda parameter. I example_match exekveras olika sektioner av kod beroende på vilken variant av VehicleEvent som tas in som argument. [3, 4]

```rust
// En datatyp/struktur "Bil".
struct Bil { gasar: bool, bromsar: bool }

// Implementationer av funktioner som finns för *alla* bilar.
impl Bil {
    fn gasar_just_nu(&self) -> bool { self.gasar }
    fn bromsar_just_nu(&self) -> bool { self.bromsar }
}

// En egenskap "Motor" så att bilen kan gasa.
trait Motor {
    fn gasa(status: &'static bool) -> &'static str;
}

// En egenskap "Bromsar" så bilen kan bromsa.
trait Bromsar {
    fn bromsa(status: &'static bool) -> &'static str;
}

// Implementerar egenskapen Motor för datatypen Bil
impl Motor for Bil {
    fn gasa(status: &'static bool) -> &'static str { /* Bilen gasar */ }
}

// Implementerar egenskapen Bromsar för datatypen Bil
impl Brosmar for Bil {
    fn bromsa(status: &'static bool) => &'static str { /* Bilen bromsar */ }
}
```

Figur 7 - Egenskaper och Strukturer

```rust
enum VehicleEvent {
    Accelerate,
    Brake,
    ShoutAt(String)
}

fn example_match(event: VehicleEvent) {
    match event {
        VehicleEvent::Accelerate => println!("Accelerating!"),
        VehicleEvent::Brake => println!("Braking!"),
        VehicleEvent::ShoutAt(person_or_thing) => println!("Shouted at {}!", person_or_thing)
    }
}

fn main() {
    let acceleration_event = VehicleEvent::Accelerate;
    let braking_event = VehicleEvent::Brake;
    let shout_at_event = VehicleEvent::ShoutAt("John".to_string());

    example_match(acceleration_event);
    example_match(braking_event);
    example_match(shout_at_event);
}
```

Figur 8 - Enumerationer

3.5. Closures

En closure (ungefär ”stängning” på svenska) i Rust är en anonym funktion som kan använda externa variabler utan speciella begränsningar. Vid användning av variabler i en closure där variabeln deklarerats utanför closuren kommer utan specificering Rust alltid använda variabler genom deras minnesreferens istället för att ta in själva värdet. Orsaken till detta beteende kommer förklaras under kapitlet ”Ägarskapsystemet” senare i avhandlingen. Ett exempel på en simpel closure finns i figur 9 nedan. [3, 4]

```
let mut a = 0;

let mut incrementer = |increment_by: i32| {
    a += increment_by;
}

incrementer(1);
incrementer(2);
```

Figur 9 - Closure

Först deklarerar vi en föränderlig variabel ”a” som innehåller heltalet 0. Därefter definierar vi en closure med namnet ”incrementer” som har en parameter ”increment_by” som modifierar variabeln ”a”. Detta resulterar i att slutgiltiga värdet av ”a” när vi nått slutet av programmet kommer vara heltalet 3. [4]

Man kan också fånga saker på basis av värde istället för referens med hjälp av nyckelordet ”move” [4]. Ett exempel på detta finns nedan i figur 10 nedan.

```
let b = 2;

let incrementer = move |increment_by: i32| {
    return b + increment_by;
}

println!(“{:?}”, incrementer(12));
println!(“{:?}”, incrementer(3));
```

Figur 10 - Closure med nyckelordet "move"

Programmet i figur 9 skulle printa ”14” följt av ”5”. Skillnaden i detta exempel är att värdet på variabel ”b” aldrig modifieras, utan man använder värdet av variabeln som om det skulle vara en konstant. I andra ord använder closuren värdet av ”b” vid kompilationstid varje gång som funktionen körs.

3.6. Felhantering

Rusts felhanteringssystem är byggt runt ett par enumerationer som är inbyggda i språket. Dessa enumerationer är *Option* och *Result*. Enumerationen Option används när det är möjligt att något endera finns eller fattas och man vill behandla de två olika fallen olikt. Option har två varianter: *Some(T)* och *None* där *T* är en generisk datatyp. Option har också en hjälpfunktion *unwrap* som enumerationer normalt inte har. Unwrap returnerar värdet i ett Option om varianten är Some, om varianten är None returneras *panic* vilket innebär att programmet kraschar. Result är väldigt lik Option, men Result har varianterna *OK(T)* och *Err(E)*, det vill säga Result brukar returneras ur funktioner på basis av om exekveringen lyckas eller misslyckas, och om exekveringen misslyckas kommer värdet inuti vara felmeddelandet. Result och Option i samarbete med ägarskapsystemet som diskuteras i nästa avsnitt tar helt bort nödvändigheten att hantera nullpekare.

En specialoperator angående felhantering i Rust är frågeteckenoperatorn (*?*) man kan placera direkt efter någon funktion som returnerar ett Result. Frågeteckenoperatorn gör så att om Result i fråga är av varianten OK, returneras värdet inuti, men om varianten är Err, returneras felet i en mer lätthanterlig form. Ett exempel på användning av frågeteckenoperatorn finns nedan i figur 11. [3, 4]

```
fn create_file() -> Result<(), std::io::Error> {
    let mut _created_file = File::create("a_test_file")?;
    Ok(())
}

fn main() {
    println!("{:?}", create_file());
}
```

Figur 11 - Frågeteckenoperatorn

4. Ägarskapsystemet

Istället för ett sophämtningssystem använder Rust en del kontroller vid kompilering. Kontrollerna eliminerar en stor del minnessäkerhetsbuggar, inklusive buffertöverfyllning, användning efter frigöring, avreferering av nullpekare och data races. Dessa kon-

troller orsakar inget extra pålägg i form av minnesanvändning eller processoranvändning. Den viktigaste gruppen av kompileringskontrollerna i Rust för att säkerställa minnessäkerhet är dess ägarskapsystem (eng. Ownership). Ägarskapsystemet består av flera komponenter som kommer beskrivas i mer detalj senare. [3, 5]

Som jämförelse använder sophämtningssystem alltid processortid för att göra sitt jobb. Detta leder till att språk med sophämtningssystem inte är lämpade för en lika stor mängd mjukvarukategorier. Sophämtningssystem orsakar trubbel i applikationer där man har till exempel stora mängder objekt som måste samlas upp. Ett konkret exempel är om man använder programmeringsspråket Java samt en objektorienterad design för att skapa ett spel, det som kan hända i en sådan situation är att man överbelastar sophämtaren vilket orsakar prestandaproblem. Dessa problem kan oftast programmeras runt, men det medföljer kompromisser man inte skulle vara tvungen att göra i icke-sophämtande språk såsom C, C++ och Rust.

Funktionen som verkställer minnessäkerhet i Rust är just dess ägarskapsystem. Ägarskaps går ut på att en variabel ha ägarskap över ett värde. Ägarskapsystemet är vad som verkligen differentierar Rust som programmeringsspråk. Ägarskapsystemet har många mindre funktionaliteter som verkställer minnessäkerhet. Ägarskapsystemet garanterar bland annat att endast en variabel har åtkomst till samma data genom att implementera ett sorts affint typsystem. Variabeln som för tillfället har åtkomst till data kallas för ägaren av datan eller bindningen till datan. Rusts ägarskapsystem tillåter endast en ägare åt gången. Ägarskapsystemet ser till att resurser som inte behövs längre, det vill säga variabler som når slutet av sin räckvidd (eng. Scope) tas bort ur minnet. Ett exempel på detta finns nedan i figur 12. Resultatet av att köra koden i figuren finns nedan i figur 13. [3, 4].

```
{
    let z = 1; // Räckvidd för variabel "z" börjar

    println!("Det fungerar! {}", z);

} // Räckvidd för variabel "z" slutar

println!("Detta fungerar inte! {}", z);
```

Figur 12 - Variabelräckvidder

```
error[E0425]: cannot find value `z` in this scope
 --> src\main.rs:8:41
  |
8 |     println!("Detta fungerar inte! {}", z);
  |                                         ^ not found in this scope
```

Figur 13 - Variabelräckvidder resultat

I Rust kan ett visst värde endast ha en ägare åt gången. Det affina typsystemet i Rust kan ses som en mer sträng version av flyttsemantik från C++. [4]

Om man skickar en variabels data i form av värdet, eller om man omdefinierar variabeln så blir destinationen den nya ägaren av datan. Det affina typsystemet säkerställer att endast denna nya plats har åtkomst till datan. Detta ser till att man inte behöver oroa sig om att annan kod modifierar datan utan explicit tillåtelse. Man behöver inte heller oroa sig om att man modifierar data som påverkar andra sektioner av kod [3,4].

Exempel på flytt av ägarskap finns nedan i figur 14**.** I figuren definierar vi en String i variabel ”aaa”. Vi flyttar värdet av variabel ”aaa” till variabel ”bbb”. Vid denna tidpunkt har variabeln ”bbb” ägarskap av värdet som tidigare ägdes av variabeln ”aaa”. Vi försöker sedan printa värdet av variabel ”aaa”, men detta går inte eftersom ”aaa” inte äger något värde för tillfället. Resultatet av detta visas nedan i figur 15. [3]

```
fn main() {
    let aaa = String::from("asd");
    let bbb = aaa;

    // ”Println-anrop 2”
    println!("{:?}", aaa);
}
```

Figur 14 - Ägarskapsflytt

```
  |
3 |     let bbb = aaa;
  |               --- value moved here
4 |
5 |     println!("{:?}", aaa);
  |                       ^^^ value borrowed here after move
  |
```

Figur 15 - Ägarskapsflytt resultat

4.1. Minnesreferenser och lån

Ett lån i Rust betyder att man använder minnesreferensen för att få åtkomst till data istället för att ta ägarskap över själva värdet. Resultatet av att använda referens istället för värde är att värdet bakom minnesreferensen inte kommer tas bort förrän själva värdet når slutet av sin räckvidd, vilket ger flexibilitet man annars inte skulle ha. [3, 4]

Ett potentiellt problem med ägarskapsystemet är att när en variabels räckvidd tar slut tas värdet i variabeln också bort. Detta kan vara ett problem till exempel om man flyttar ägarskapet av en variabel till en ny variabel i en funktion, och den inre variabelns räckvidd tar slut. I detta scenario får man samma felmeddelande som ovan i figur 15 eftersom man försöker använda ett värde vars ägare inte längre finns i rätt räckvidd. [3]

Rusts lösning på tidigare nämnda scenario är att skapa en underkomponent till ägarskapsystemet som kallas ”the borrow-checker” (ungefär ”lånekollaren” på svenska). Lånekollarens enda uppgift är att se till att alla lån i mjukvaran är giltiga. För att låna ett värde istället för att ta ägarskap använder man symbolen ”&”. Ett exempel på användning av detta finns nedan i figur 16. [3]

```
fn main() {
    let a = String::from("asd");
    do_something(&a);
    do_something(&a);
}

fn do_something(b: &String) {
    println!("{}", b);
}
```

Figur 16 - Lån av värden

I exemplet har vi en funktion ”do_something” som tar emot en *referens* ”b” som argument. I ”main” använder vi ”do_something” genom att skicka in minnesreferensen till ”a”. Det är också möjligt att ändra på värdet bakom en minnesreferens genom att byta ut alla ampersand till ”&mut”, som nedan i Figur 17. [3]

```rust
fn main() {
    let mut a = String::from("asd");
    do_something(&mut a); // Skriver ut "asd123"
    do_something(&mut a); // Skriver ut "asd123123"
}

fn do_something(b: &mut String) {
    b.push_str("123");
    println!("{}", b);
}
```

Figur 17 - Föränderligt lån av värden

4.2.Livstider

En viktig komponent av ägarskapsystemet i Rust är livstider (eng. Lifetimes). Livstidssystemet tar inspiration från bland annat språket Cyclone som implementerade ungefär samma sak men kallade funktionaliteten ”regioner”. Livstider i Rust används av kompilatorn för att se hur länge minnesreferenser(variabler) är giltiga. Livstiden av en variabel identifieras med hjälp av en enkel apostrof, till exempel *’a*. [3, 4]

Såsom variabler har livstider också räckvidder. En illustration av dessa räckvidder finns nedan i figur 18. I figuren finns en variabel *a* som lånar värdet ur variabeln *b*. I figuren illustreras tydligt var olika livstider är giltiga. I figuren illustreras räckvidderna för livstiden av variablerna a och b, vars livstider heter ’a och ’b respektive.

```rust
fn main() {
    let a = "asd";      // -----------+-- 'a
                        //            |
    let b = &a;         // --+-- 'b   |
                        //   |        |
    println!("{}", b);  //   |        |
                        // --+        |
}                       // -----------+
```

Figur 18 - Livstidsräckvidder

Livstidssystemet i Rust har en funktionalitet där man kan explicit specificera relationen mellan minnesreferenser. Rust försöker alltid automatiskt se var minnesreferensernas livstider börjar, men denna automatisering är inte tillräcklig i alla situationer, i vissa situationer måste man explicit definiera livstider på variabler för att lånekollaren skall ha tillräcklig information för att fungera korrekt. Ett exempel på en situation där det

inte räcker med automatisk livstidsupptäckt finns nedan i figur 19**Error! Reference source not found.**.

```
fn main() {
    let intx: i32 = 37;
    let inty: i32 = 12;

    let result = bigger(&intx, &inty);
}

fn bigger(a: &i32, b: &i32) -> &i32 {
    if(a > b) {
        a
    } else {
        b
    }
}
```

Figur 19 - Tanke bakom explicit livstidsdeklaration

Koden i **Error! Reference source not found.** kommer inte kompileras, utan man kommer bli given en text som förklarar att kompilatorn inte utan hjälp kan identifiera om argument *a* eller *b* kommer returneras ur metoden *bigger*. Detta skulle inte vara ett problem om kompilatorn skulle veta hur livstiderna av *a* och *b* relaterar till livstiden av värdet som returneras. För att lösa detta problem kan vi modifiera funktionen *bigger* så att den ser ut som i figur 20 nedan. [3]

```
fn bigger<'a>(a: &'a i32, b: &'a i32) -> &'a i32 {
    if(a > b) {
        a
    } else {
        b
    }
}
```

Figur 20 - Explicit livstidsdeklaration

Ändringen som gjordes var att lägga till *'a* som generisk parameter i funktionen genom att skriva <*'a*> direkt efter funktionsnamnet. Nu är också *'a* explicit specificerad som livstiden för båda parametervariablerna samt returvärdet. I andra ord säger funktionsdeklaration åt kompilatorn att livstiden av *bigger*s returvärdes livstid kommer vara åtminstone lika länge som *'a*, där *'a* är den längre livstiden av de två input-argumenten. Alternativt kan man specificera en livstid per input-parameter, vilket tillåter returvärdet att använda någondera av livstiderna. Att använda explicita livstidsannoteringarna i

funktionsdeklarationen modifierar man inte hur länge de olika värdenas livstider faktiskt varar, utan man hjälper endast kompilatorn förstå hur ett flertal minnesreferenser hör ihop. [3]

Vissa programmeringsmönster som används med livstider är så vanliga att kompilatorn i vissa situationer implicit lägger till livstidsparametrarna. Denna automatisering kallas livstidselision. Ett exempel som visar skillnaden mellan explicit livstidsdefiniering och elision finns nedan i figur 21**Error! Reference source not found.**. Bägge funktionerna kompileras till identisk maskin kod.

```
fn explicit_lifetime<'a>(x: &'a i32) {
    println!("{}",x)
}

fn implicit_lifetime(x: &i32) {
    println!("{}",x)
}
```

Figur 21 - Livstidselision

5. Trådar

Rust kombinerar alla tidigare diskuterade ämnen för att skapa ett minnessäkert multitrådningssystem som är enkelt att använda. Rusts multitrådningssystem använder ”1-till-1” multitrådningsmodell, dvs varje tråd man skapar i Rust representeras av en tråd i operativsystemet. Fastän Rust använder denna multitrådningsmodell finns det bibliotek tillgängliga som möjliggör användningen av en ”M-till-N” multitrådningsmodell. Motivationen för ”1-till-1” multitrådning i Rust är den att man vill bli av med allt pålägg i systemutvecklingsspråk. En tråd i Rust skapas genom att använda funktionen *spawn* i biblioteket *std::thread.* Spawn har en enda parameter, en closure, som kommer exekveras av tråden. Denna closure stöder på samma sätt som vanliga closures nyckelordet move. Move behövs speciellt mycket när man arbetar med trådar eftersom man inte kan veta på förhand hur länge en tråd kommer köra. Detta ovetande innebär att minnesreferenser från utanför closuren kan nå slutet av sin livstid före trådens exekvering är färdig, vilket inte är kompatibelt med minnessäkerhet och orsakar att programmet inte kompileras. [3, 4]

Den tidigare nämnda funktionen spawn returnerar ett värde av datatypen JoinHandle. Värden av typen JoinHandle har en funktion join() vars exekvering orsakar alla eventuella andra trådar att vänta på att trådens exekvering är färdig. Nedan i figur <**int**> finns ett exempel där en tråd skapas samt funktionerna spawn och join används. I denn kodsnutt väntar huvudtråden på att tråden bakom variabeln *join_handle* ska exekvera färdigt före programmet i sin helhet har kört färdigt. I figur 22 nedan skapas en tråd vars JoinHandle sparas i variabeln join_handle. Före huvudtrådens exekvering är färdig väntar den på att den andra trådens exekvering klargjorts. [3, 4]

```
let join_handle = thread::spawn(|| {
    println!("From within a spawned thread!");
});

println!("From the main thread!");

join_handle.join().unwrap();
```

Figur 22 - Grundläggande multitrådning

Rust använder så kallad *meddelandesändning* (eng. Message Passing) för att minnessäkert skicka data mellan olika trådar. Meddelandesändning i Rust implementeras med s.k. kanaler (eng. channels). Varje kanal har en eller flera sändare samt specifikt en mottagare, ofta benämnda *tx* och *rx*. Om sändaren eller mottagaren tappar anslutningen till sin motpart, stängs kanalen tills det finns mer information att ta emot för mottagaren. Sändarparten, det vill säga variabeln tx, har en funktion *send(T) -> Result(T, E)* som används för att skicka data till mottagaren. Mottagaren, det vill säga variabeln rx har en funktion *recv() -> Result(T, E)* som används för att ta emot en sändning. Mottagaren kan användas som en iterator, vilket är användbart om det kan finnas flera inkommande meddelanden man vill ta emot. Använder man mottagaren som iterator kommer kanalen stängas först när man bearbetat alla inkommande meddelanden. Använder man *recv().unwrap()* utan inkommande information som väntar, får huvudtråden panik på grund av *RecvError*. Nedan i figur 23 skapas en kanal som har två sändare genom kloning av den första sändaren *tx*. Tråden som skapas med hjälp av funktionen spawn tar ägarskap av bägge sändare, dessa sändare skickar ett par meddelande som tas emot på huvudtråden. [3, 4]

```
let (tx, rx) = channel();
let tx2 = tx.clone();

thread::spawn(move || {
    tx.send("Hejsan!").unwrap();
    thread::sleep(Duration::from_millis(1000));
    tx2.send("Och hejdå!").unwrap();
});

for data in rx {
    println!("Received data:{}", data);
}
```

Figur 23 - Multitrådning med kanaler

6. Avslutning

(Inte skriven ännu)

7. Figurer

8. Källor

[1] P. Bright, ”How security flaws work: The buffer overflow”, Ars Technica, Augusti 26, 2015. [Online]. Tillgänglig: https://arstechnica.com/information-technology/2015/08/how-security-flaws-work-the-buffer-overflow/. [Hämtad Februari 18, 2004].

[2] W. Xu, D. C. DuVarney och R. Sekar, ”An efficient and backwards-compatible transformation to ensure memory safety of C programs”, ACM SIGSOFT Software Engineering Notes, vol. 29, pp. 117-126, Dec. 1965.

[3] S. Klabin och C. Nichols, ”The Rust Programming Language”, California: No Starch Press, 2018.

[4] A. Beingessner, ”*You can’t spell trust without Rust*”, Carleton University, Ottawa, Ontario, Canada, 2015. [Online] Tillgänglig: https://raw.githubusercontent.com/Gankro/thesis/master/thesis.pdf

[5] J. Blandy och J. Orendorff, ”Programming Rust”, Sebastopol, California: O’Reilly Media, 2017.

[6] J. Blandy, ”Why Rust?”, Sebastopol, California: O’Reilly Media, 2015.