Key Technology in Trusted Programming

Department：Trusted PLDI Lab / Ireland Research Center / European Research Institute
Author：Professor Yijun Yu (Huawei Ireland and The Open University UK)
on behalf Huawei's MemSafePro ITMT project members and Rust community experts
Date：July 7, 2022 @ 2012 Labs Global Software Technology Summit
Version：1.0

Trusted Programming we develop enhance the efficiency and the proficiency of new programming technology towards high productivity in creating high quality software, which guarantee safety and performance.

HUAWEI

Optimize CRustS to increase the proportion of safe code and check temporal safety issues.

Requirements

To ensure safety, the C code needs to be migrated to the Rust code. Due to the huge legacy of C code in product lines and the steep learning curve of the Rust language, an automated C-to-Rus code conversion tool was developed to assist in manual secure code migration.

Target Product

Product lines with existing C codes and requirements for converting to Rust codes.

Dependencies

The target C code can be compiled normally in the original environment and architecture.

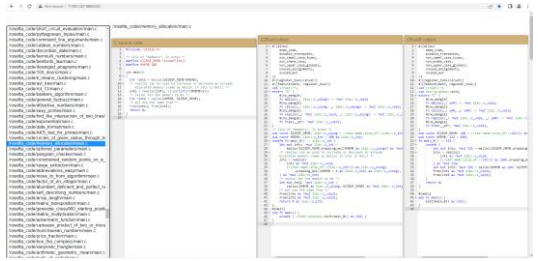
Improving raw pointer to reference refactoring since phase 1:

- ✓ Enhance the detection capability of time memory safety vulnerabilities for C code that could not be detected by Microsoft Checked C.
- ✓ For Rust code that can be translated resulting in a higher safety code ratio:
  - ✓ By reducing the number of unsafe regions involving raw pointer expressions

Memory safety Issues	NIST Class	No. of Samples	Detected	Ratio
Memory leakage	CWE401	612	234	38.2%
repeated release	CWE415	336	156	46.4%
dangling pointer	CWE416	150	74	49.3%
Heap Buffer Overflow	CWE122	3656	631	17.2%

NIST Juliet 1.3 2017 Dataset C/C++ Memory safety Detection, while Microsoft Checked C could not check

- Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James Cordy, Ahmed Hassanl. " In Rust we Trust: Translating Unsafe C to Safer Rust", ICSE 2022.
- Emre, Schroeder, Dewey, Hardekopf, "Translating C to Safer Rust", In: OOPSLA, 2021.
- Yamaguchi, Matsuda, David, Wang, "Synbit: Synthesizing Bidirectional Programs using Unidirectional Sketches", In OOPSLA 2021.

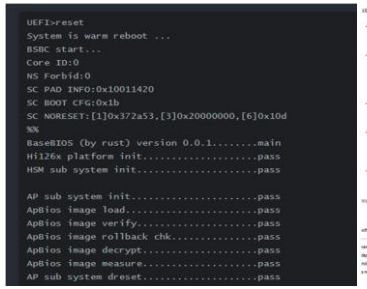


A playground has been deployed in Huawei.

Code has been released to intersource community

Verify 100% correct and enabled development.

BaseBIOS of Central Hardware



Wireless Q922 Module

- The innovative method solves the raw pointer to reference refactoring transformation, breaks through the original TXL ceiling, and increases the safe code ratio to 53%, surpassing [OOPSLA21] and our Phase 1 [ICSE22] work (40%). In addition, innovative check analysis algorithms are used to detect time-related memory safety issues. The CWE401, CWE415, CWE416, and CWE122 reach 17% to 49% respectively, surpassing Microsoft Checked C.
- Develop the Playground demos, which helps the experience and promotion of Rust in the company, and released code and binary to the intersource community to enable C/C++ product line to switch to Rust.
- Wireless Q922 module, BaseBIOS base software pilot and board verification, and HiSilicon Kirin platform xloader: C-to-Rust code conversion functions are complete and correct, which are auxiliary and reference for pilot projects.

HiSilicon Kirin xLoader has been put into commercial use with Harmony 3.0 phones.

Kirin 启动 firmware xLoader 的安全校验模块 Rust 重构项目，该项目是团队内首次使用 Rust 进行产品模块重构的项目，面临人员能力导入困难，项目采用使用 C2Rust 工具对待重构代码进行转译、然后再对转译的 Rust 代码进行优化，极大提升了首次使用 Rust 编码的开发效率；项目组也与 C2Rust 工具团队一起进行了总结，对工具后续优化提供了若干思路。目前该项目已经完成交付，随鸿蒙 3.0 推送手机产品商用。

Rust: an efficient language designed to be safe

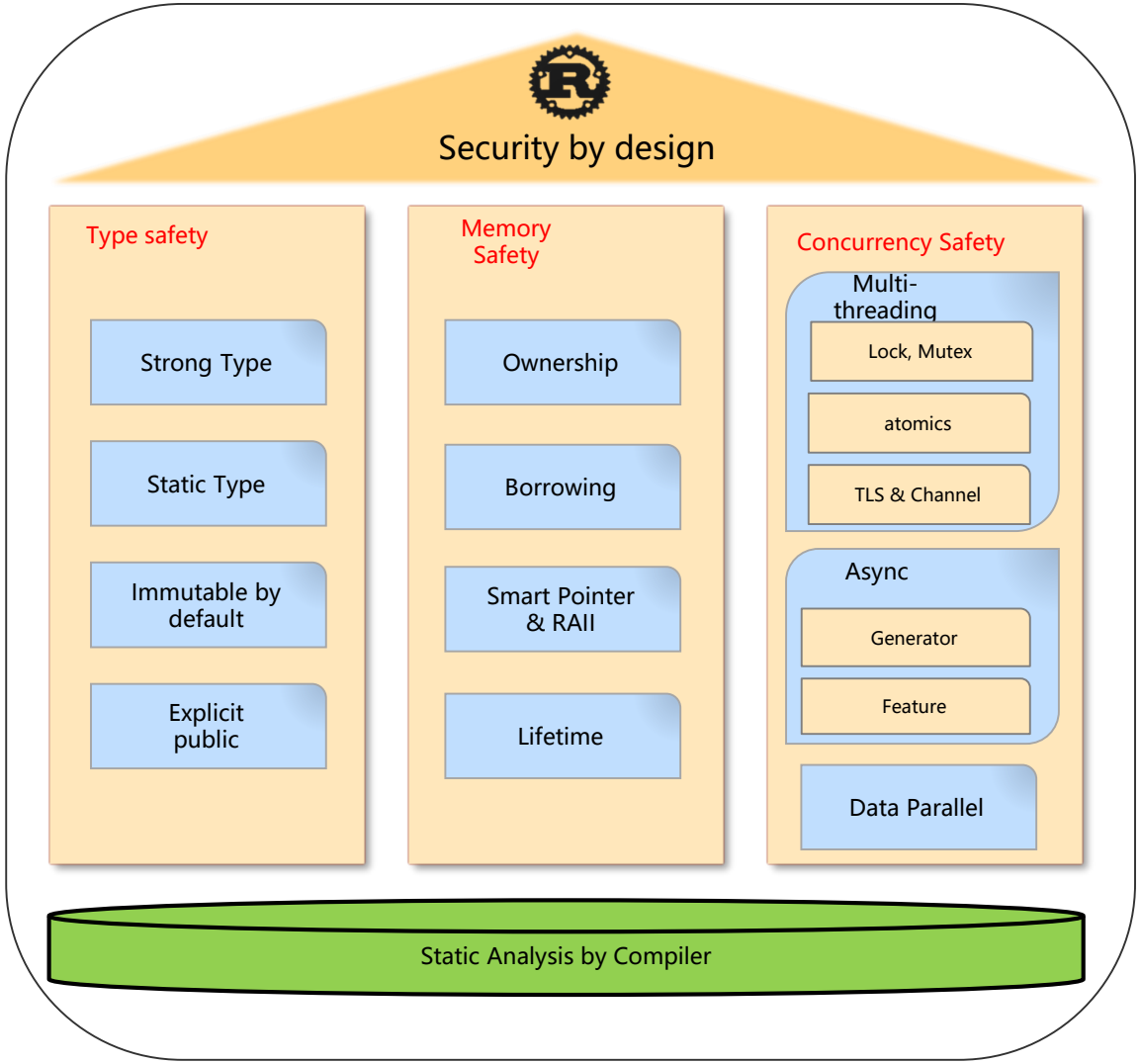


Figure 1. Rust History. *(Each blue point shows the number of feature changes in one release version. Each red point shows total LOC in one release version.)*

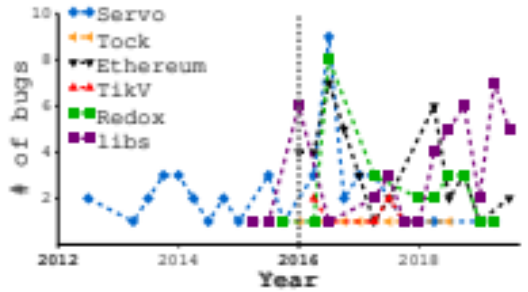


Figure 2. Time of Studied Bugs. *(Each point shows the number of our studied bugs that were patched during a three month period.)*

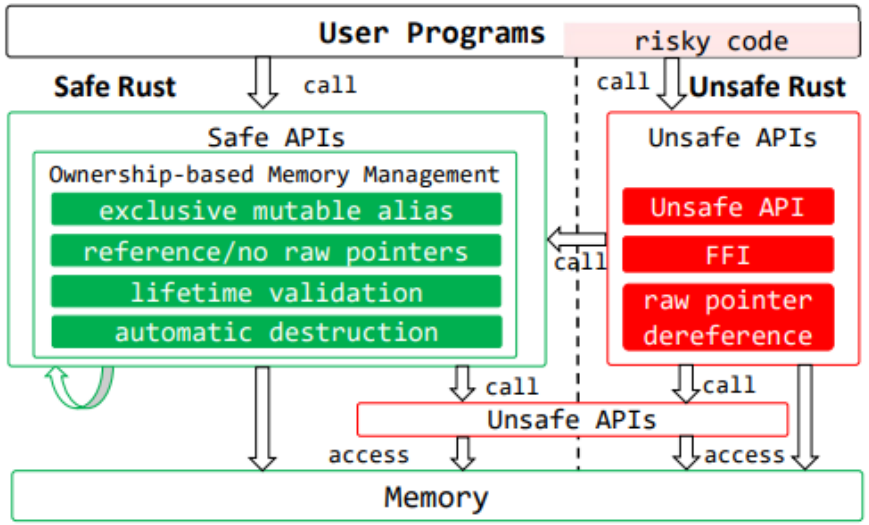


Figure 1: Idea of Rust for preventing memory-safety bugs.

Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs

Boqin Qin* BUPT, Pennsylvania State University USA
Yilun Chen† Purdue University USA
Zeming Yu Pennsylvania State University USA
Linhai Song Pennsylvania State University USA
Yiying Zhang University of California, San Diego USA

Some Rust projects inside Huawei

- **Wireless TLV encodec framework (3K LOC)**
  - > No memory safety issues, hard to have runtime errors
  - > Performance speed up 4~7 times
  - > Learning curve 4 weeks, Memory consumption increases from C to Rust (532K→696K when there are 161 fields)
- **Central Soft OpenEuler StratoVirt (24K LOC)**
  - > Zero vulnerabilities, No memory safety issues (passed ICSL tests)
  - > Learning curve 4 weeks (initial) till 6 months (master)
- **Central Soft Light-weight container (20K LOC)**
  - > No memory safety issues
  - > Learning curve: 4 weeks (initial) till 6 months (master)
  - > Learning through excellent open-source projects
- **5G Core UDG Flow Module (11K LOC)**
  - > No Memory Safety issues
  - > Memory, CPU consumptions and performance level off to C
  - > Learning curve 2~4 weeks
- **Hisilicon Camera Dirt Detection algorithm (8K LOC)**
  - > No memory safety issues
  - > Performance enhances to 25% (2.004 -> 1.603 ms/frame)
  - > Memory fluctuation reduces by 50%
  - > Learning curve 2~4 weeks
  - > Shared library (SO) size (97K->107K)
- **Ylong-Rust (100 KLOC), available on OpenX**
  - > No memory safety issues
  - > Async concurrency framework, Crypto, JSON/XML parsing, hot patches, etc..
  - > Learning curve 2~4 weeks
  - > 1 old·+··1 new paired programming for 2 months ·5K LOC/per person
- **Central Hardware BaseBIOS (6K LOC)**
  - > Support ARM Cortex, M and RISC-V MCU hardware abstraction
  - > Learning curve 3~5 weeks, pure embedded bare metal with no-std
  - > CRustS converts Rust code can compile and run with good efficiency, but some manual adjustments needed for commits

Harmony and Open Harmony: Network Management modules

Our Vision of Trustworthy System Programming (Rust): *Automated* Programming for Better *Safety*, *Performance*, and *Productivity*

Theoretical Foundations of Trusted Programming Research

Memory Safety = Zero-cost Memory Abstraction
Fearless Concurrency = Lambda Calculus for FP
+ Neural Network Representations of Code

Challenges in Trusted Programming for Huawei Systems

- Legacy C/C++ code:
  LOC >80%, Vulnerabilities >50%
- Developing new systems in Rust:
  - Steep learning curve: 2-4 months
  - Undefined behaviour in unsafe code
- Low code model-driven development:
  Sanyapo battle: lack debugging capabilities for meta-programming

Technology breakthroughs

- Code transformation of safer Rust code【ICSE22】
  Commercial development in BaseBIOS (CH)、xloader (Hisilicon)【Harmony OS 3.0】
- Rust Roadmap contributions: Async/Await、Futures
  Inline Assembly, Cross Compilation [Wireless BU】
  Crypto、SIMD libraries【Camera of Hisilicon】
  Fearless Concurrency【Scoped Threads，Mutex, FFTS】
- Bidirectional Transformations of MDD【ICSE'12】
  InferCode Pre-training【ICSE21】
  Corder: Cross SE tasks ML【AAAI'21, SIGIR21】

5 Top-Conference publications，2 Patents

Impact

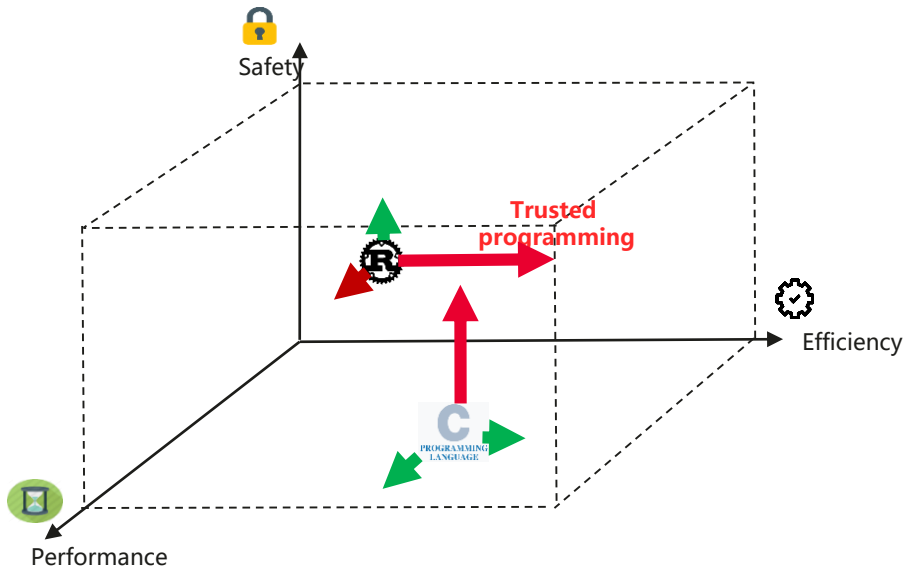
- Auto migration of systems in 3 days
  with safe ratio 60% and
  no performance penalty from C 100%
- Made 9 Rust roadmap contributions to language, compiler, standard library;
- Accurate unsafe Rust classification 98%
  classify assembly code accuracy 77%;
  Performance boost of Camera applications 25%
  Lock performance >2x
- Reduce Open Euler Rust toolchain build bottleneck by 3.5 times.

Matching Strategic Innovation Project (STIP) needs in support for the ITMT project《MemSafePro》:

Recruited top Rust experts from Rust Foundation and community leading teams from Europe:
Amanieu d'Antras (Standard library, UK）, Mara Bos（Rust Library Co-Leader, Nederland）, Guillaume Gomez (DevTools, France),
Vadim Petrochenkov (Language, Russia), David Wood (Compiler, UK) and Bastian Kauschke（Compiler, Germany）

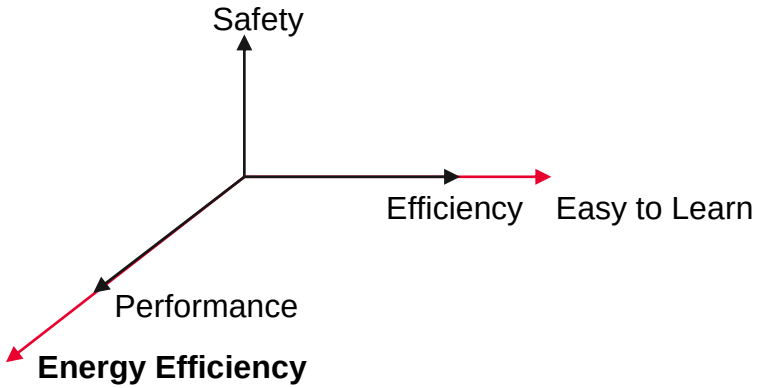
Collaboration with top universities world-wide:
The Open University UK – Helen Sharp《Bug Localization》, Bristol University UK -- Meng Wang《eDSL Debugging》 Cristina David《C2Rust Program Synthesis》（EPSRC、Royal Society)
Lero Ireland -- Bashar Nuseibeh《RE4AI》, SMU Singapore -- Lingxiao Jiang《Deep Learning of Code》
Peking University China -- Zhenjiang Hu《FP Core semantics for C and Rust》, Yingfei Xiong《Beyond Alpha Code – Synthesis Rust Algorithms》

Mobile Rust：Safe Programming for Mobile Ecosystem with Fearless Concurrency and High Productivity



- **Virtues：Safe, Efficient, Productive**
  - Memory Safety, Concurrency Safety、Thread Safety
  - As performant as C/C++
  - More efficient and maintainable than C/C++
- **Challenges：High Learning Curve**
  - Borrowed many language features
  - Share memory model between developer and compiler

Mobile Rust



- Migrate C/C++ modules
- Build open programming ecosystem to differentiate Swift and Kotlin

Performance Tuning for Mobile hardware platform

- Hardware friendly performance tuning for FFTS
- Cross-compilation for new processors
- Compiler optimization for LLVM backend
- Modern concurrency programming paradigms for IO/CPU-intensive apps
- Memory hierarchy optimizations

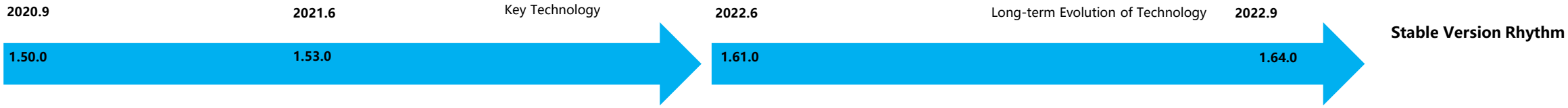
Lowering the learning curve and striking a balance between usability and software performance

- Dynamic link for better reusability
- Interoperability between FFI languages
- Promote usability by simplifying Rust to interoperate with Rust toolchains
- DSL at the app-development level to interoperate with Java/kotlin/dart/swift。

Huawei Making Contributions to Complete 9 Roadmap Features of Rust Community in 2022

Contributed to the roadmap features that are valued important by Huawei's product lines

Roadmaps	Features	Description	Value to Huawei	Engagement	Completion status (commercial ready version)
Compiler	Inline Assembly	From Experimental to Stable	BaseBIOS advanced safety module	**Driving**	**1.59.0**
Language	FFI unwind	Resolve outstanding issues of cross-language interoperability	FFI between DOPRA Rust and DOPRA C	**Participating**	**1.64.0 (as planned)**
Compiler	Cross compilation	Formal tiered policy to support compilation objectives	BaseBIOS advanced safety module of Central Hardware and HiSilicon Kirin xloader reconstruction	**Driving**	**1.59.0**
Compiler	Faster compilation	Compile speed 20% - 30% faster	Improves Compilation and Build Speed	**Driving**	**1.61.0**
Language	Uninitialized Buffer Processing	The I/O buffer is not initialized.	Enhances Rust Compiler safety	**Participating**	**1.64.0 (as planned)**
Standard Library	std::arch	SIMD standard library support	Refactors of HiSilicon Graphic Camera Algorithm	**Driving**	**1.59.0**
Standard Library	std::sync	Support for Mutex, RwLock and CondVar on all platforms	Improves concurrency and lock performance	**Driving**	**1.63.0**
Standard Library	std::error	Simplify error handling API	Improves the usability of the debugging interface	**Participating**	**1.60.0**
DevTools	Rustdoc Navigation	Cross-Document Navigation Link Generation	Improves the E2E tool chain	**Driving**	**1.61.0**
Standard Library	Scoped Threads	Supports syntax extension of multithreaded concurrent code.	Improves the development of fearless concurrency	**Driving**	**1.61.0**
Compiler	Split DWARF	Remove debugging information from binaries	Reduces binary file size	**Driving**	**1.61.0**
Compiler	Top Issues of LLVM affecting Rust compiler	Stability Improvement based on LLVM Compilation	Resolves three blockages	**Driving**	**LLVM15**



Quicksort in C and C++

https://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Rust

```c
void quicksort(int *A, int len) {
  if (len < 2) return;

  int pivot = A[len / 2];

  int i, j;
  for (i = 0, j = len - 1; ; i++, j--) {
    while (A[i] < pivot) i++;
    while (A[j] > pivot) j--;

    if (i >= j) break;

    int temp = A[i];
    A[i]     = A[j];
    A[j]     = temp;
  }

  quicksort(A, i);
  quicksort(A + i, len - i);
}
```

```cpp
#include <iterator>
#include <algorithm> // for std::partition
#include <functional> // for std::less

template<typename RandomAccessIterator,
         typename Order>
 void quicksort(RandomAccessIterator first, RandomAccessIterator last, Order order)
{
  if (last - first > 1)
  {
    RandomAccessIterator split = std::partition(first+1, last, std::bind2nd(order, *first));
    std::iter_swap(first, split-1);
    quicksort(first, split-1, order);
    quicksort(split, last, order);
  }
}

template<typename RandomAccessIterator>
 void quicksort(RandomAccessIterator first, RandomAccessIterator last)
{
  quicksort(first, last, std::less<typename std::iterator_traits<RandomAccessIterator>::value_type>());
}
```

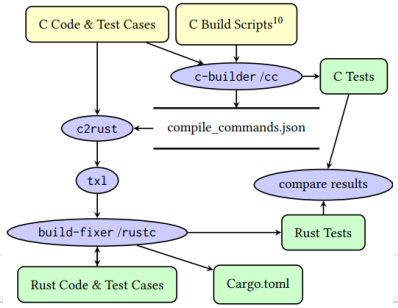
Generic，Polymorphism

Algorithm	Safety	Performance	Productivity

Quicksort in Rust

https://c2rust.com/
https://185.190.206.130

Ling, Michael; Yu, Yijun; Wu, Haitao; Wang, Yuan; Cordy, James and Hassan, Ahmed (2022). CRustS: A Transpiler from Unsafe C to Safer Rust. In: *44th International Conference on Software Engineering (ICSE 2022)*, 22-27 May 2022, Pittsburgh, USA.



Not secure | 185.190.206.130

Select a C code: **edit or upload your own C code:** **you will see output Rust code from C2Rust:** **and CRustS:**

./rosetta_code/determine_if_a_string_has_all_
./rosetta_code/cut_a_rectangle/main.c
./rosetta_code/hilbert_curve/main.c
./rosetta_code/magic_8_ball/main.c
./rosetta_code/check_output_device_is_a_ter
./rosetta_code/memory_allocation/main.c
./rosetta_code/crc32/main.c
./rosetta_code/cartesian_product_of_two_or_r
./rosetta_code/mandelbrot_set/main.c
./rosetta_code/arbitrary_precision_integers/ma
./rosetta_code/horners_rule_for_polynomial_e
./rosetta_code/next_highest_int_from_digits/m
./rosetta_code/emirp_primes/main.c
./rosetta_code/matrix_multiplication/main.c
./rosetta_code/null_object/main.c
./rosetta_code/nim_game/main.c
./qsort/qsort.c
./libxml2/relaxng.c
./libxml2/SAX2.c
./libxml2/entities.c
./libxml2/xmlregexp.c
./libxml2/xmlcatalog.c
./libxml2/dict.c
./libxml2/encoding.c
./libxml2/xmlstring.c
./libxml2/threads.c
./libxml2/pattern.c
./libxml2/globals.c
./libxml2/buf.c

C Source of ./qsort/qsort.c

```c
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

int partition (int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high)
{
    if (low < high) {
        int i = partition(arr, low, high);
        quickSort(arr, low, i - 1);
        quickSort(arr, i + 1, high);
    }
}
```

Upload | Translate | Query

C2Rust output of ./qsort/qsort.c

```rust
// Safe API ratio of c2rust is: 0 %
// Safe code ratio of c2rust is: 100 %
#![allow(dead_code, mutable_transmutes, non_camel_case_types, non_snake_case,
         non_upper_case_globals, unused_assignments, unused_mut)]
#![register_tool(c2rust)]
#![feature(register_tool)]
#[no_mangle]
pub unsafe extern "C" fn swap(mut a: *mut libc::c_int,
                              mut b: *mut libc::c_int) {
    let mut t: libc::c_int = *a;
    *a = *b;
    *b = t;
}
#[no_mangle]
pub unsafe extern "C" fn partition(mut arr: *mut libc::c_int,
                                   mut low: libc::c_int,
                                   mut high: libc::c_int) -> libc::c_int {
    let mut pivot: libc::c_int = *arr.offset(high as isize);
    let mut i: libc::c_int = low - 1 as libc::c_int;
    let mut j: libc::c_int = low;
    while j <= high - 1 as libc::c_int {
        if *arr.offset(j as isize) <= pivot {
            i += 1;
            swap(&mut *arr.offset(i as isize), &mut *arr.offset(j as isize));
        }
        j += 1
    }
    swap(&mut *arr.offset((i + 1 as libc::c_int) as isize),
         &mut *arr.offset(high as isize));
    return i + 1 as libc::c_int;
}
#[no_mangle]
pub unsafe extern "C" fn quickSort(mut arr: *mut libc::c_int,
                                   mut low: libc::c_int,
                                   mut high: libc::c_int) {
    if low < high {
        let mut i: libc::c_int = partition(arr, low, high);
        quickSort(arr, low, i - 1 as libc::c_int);
        quickSort(arr, i + 1 as libc::c_int, high);
    };
}
```

Download c2rust.rs | Refactor | Query

CRustS output of ./qsort/qsort.c

```rust
// Safe API ratio of crusts is: 100 %
// Safe code ratio of crusts is: 0 %
#![allow (dead_code, mutable_transmutes, non_camel_case_types, non_snake_case,
#![register_tool (c2rust)]
#![feature (register_tool)]
#[no_mangle]
pub extern "C" fn swap (mut a : * mut i32, mut b : * mut i32) {
    unsafe {
        let mut t : i32 = * a;
        * a = * b;
        * b = t;
    }
}

#[no_mangle]
pub extern "C" fn partition (mut arr : * mut i32, mut low : i32, mut high : i
    unsafe {
        let mut pivot : i32 = * arr.offset (high as isize);
        let mut i : i32 = low - 1;
        let mut j : i32 = low;
        while j <= high - 1 {
            if * arr.offset (j as isize) <= pivot {
                i += 1;
                swap (& mut * arr.offset (i as isize), & mut * arr.offset (j a
            }
            j += 1
        }
        swap (& mut * arr.offset ((i + 1i32) as isize), & mut * arr.offset (hi
        return i + 1;
    }
}

#[no_mangle]
pub extern "C" fn quickSort (mut arr : * mut i32, mut low : i32, mut high : i
    unsafe {
        if low < high {
            let mut i : i32 = partition (arr, low, high);
            quickSort (arr, low, i - 1);
            quickSort (arr, i + 1, high);
        };
    }
}
```

Download crusts.rs | Query | Predict Unsafe

Quicksort in Rust (manual)

https://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Rust

```
fn quick_sort<T,F>(v: &mut [T], f: &F)
    where F: Fn(&T,&T) -> bool
{
    let len = v.len();
    if len >= 2 {
        let pivot_index = partition(v, f);
        quick_sort(&mut v[0..pivot_index], f);
        quick_sort(&mut v[pivot_index + 1..len], f);
    }
}

fn partition<T,F>(v: &mut [T], f: &F) -> usize
    where F: Fn(&T,&T) -> bool
{
    let len = v.len();
    let pivot_index = len / 2;
    let last_index = len - 1;

    v.swap(pivot_index, last_index);

    let mut store_index = 0;
    for i in 0..last_index {
        if f(&v[i], &v[last_index]) {
            v.swap(i, store_index);
            store_index += 1;
        }
    }

    v.swap(store_index, len - 1);
    store_index
}
```

In-place swap

```
fn quick_sort<T, E>(mut v: T) -> Vec<E>
where
    T: Iterator<Item = E>,
    E: PartialOrd,
{
    match v.next() {
        None => Vec::new(),

        Some(pivot) => {
            let (lower, higher): (Vec<_>, Vec<_>) = v.partition(|it| it < &pivot);
            let lower = quick_sort(lower.into_iter());
            let higher = quick_sort(higher.into_iter());
            lower.into_iter()
                .chain(core::iter::once(pivot))
                .chain(higher.into_iter())
                .collect()
        }
    }
}
```

Or, using functional style (slower than the imperative style but faster than functional style in other languages):

Quicksort in Rust

https://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Rust

```
fn quick_sort<T,F>(v: &mut [T], f: &F)
    where F: Fn(&T,&T) -> bool
{
    let len = v.len();
    if len >= 2 {
        let pivot_index = partition(v, f);
        quick_sort(&mut v[0..pivot_index], f);
        quick_sort(&mut v[pivot_index + 1..len], f);
    }
}

fn partition<T,F>(v: &mut [T], f: &F) -> usize
    where F: Fn(&T,&T) -> bool
{
    let len = v.len();
    let pivot_index = len / 2;
    let last_index = len - 1;

    v.swap(pivot_index, last_index);

    let mut store_index = 0;
    for i in 0..last_index {
        if f(&v[i], &v[last_index]) {
            v.swap(i, store_index);
            store_index += 1;
        }
    }

    v.swap(store_index, len - 1);
    store_index
}
```

Mutable Reference Limits Parallelism

1) Borrow checker can tell the variable may overlap
2) Refactoring of the &mut to & would allow the parallelization
3) clone the variable could allow for parallelization
4) scoped thread may still parallelize it with ease
clone the slice and join the threads.

Special transformation may be designed as a compiler dataflow pass.

In-place swap

Safe Rust makes it easier to achieve parallelization

```
fn quick_sort<T, E>(mut v: T) -> Vec<E>
where
    T: Iterator<Item = E>,
    E: PartialOrd,
{
    match v.next() {
        None => Vec::new(),

        Some(pivot) => {
            let (lower, higher): (Vec<_>, Vec<_>) = v.partition(|it| it < &pivot);
            let lower = quick_sort(lower.into_iter());
            let higher = quick_sort(higher.into_iter());
            lower.into_iter()
                .chain(core::iter::once(pivot))
                .chain(higher.into_iter())
                .collect()
        }
    }
}
```

No dependency between left and right;

Recursive function can be parallelized using FFTS.

Or, using functional style (slower than the imperative style but faster than functional style in other languages):

How do we parallelize sequential QuickSort in Rust?

1. Scoped Threads (available in nightly Rust, will be stablized in 1.61.0)

```
fn quick_sort<T,F>(v: &mut [T], f: &F)
    where F: Fn(&T,&T) -> bool
{
    let len = v.len();
    if len >= 2 {
        let pivot_index = partition(v, f);
        quick_sort(&mut v[0..pivot_index], f);
        quick_sort(&mut v[pivot_index + 1..len], f);
    }
}

fn partition<T,F>(v: &mut [T], f: &F) -> usize
    where F: Fn(&T,&T) -> bool
{
    let len = v.len();
    let pivot_index = len / 2;
    let last_index = len - 1;

    v.swap(pivot_index, last_index);

    let mut store_index = 0;
    for i in 0..last_index {
        if f(&v[i], &v[last_index]) {
            v.swap(i, store_index);
            store_index += 1;
        }
    }

    v.swap(store_index, len - 1);
    store_index
}
```

```
1   #![feature(null, scoped_threads)]
2   use std::thread;
3
4   fn quick_sort<T,F>(v: &mut [T], f: &F)
5       where F: Fn(&T,&T) -> bool
6   {
7       let len = v.len();
8       if len >= 2 {
9           let pivot_index = partition(v, f);
10          let (lower, rest) = v.split_at_mut(pivot_index);
11          let (mid, higher) = rest.split_at_mut(1);
12          thread::scope(|s| {
13              s.spawn(|| {
14                  quick_sort(lower, f); // Executed by a new thread.
15              });
16              quick_sort(higher, f); // Executed by the original thread.
17          });
18      }
19  }
```

Performance：1024*1024 Array random numbers，

1100 ms （Single thread）

37 ms （Multi-threaded）

2. Rayon library

https://docs.rs/rayon/latest/rayon/fn.join.html

```rust
fn quick_sort<T:PartialOrd+Send>(v: &mut [T]) {
   if v.len() > 1 {
       let mid = partition(v);
       let (lo, hi) = v.split_at_mut(mid);
       rayon::join(|| quick_sort(lo),
                   || quick_sort(hi));
   }
}

// Partition rearranges all items `<=` to the pivot
// item (arbitrary selected to be the last item in the slice)
// to the first half of the slice. It then returns the
// "dividing point" where the pivot is placed.
fn partition<T:PartialOrd+Send>(v: &mut [T]) -> usize {
    let pivot = v.len() - 1;
    let mut i = 0;
    for j in 0..pivot {
        if v[j] <= v[pivot] {
            v.swap(i, j);
            i += 1;
        }
    }
    v.swap(i, pivot);
    i
}
```

Conceptually, calling `join()` is similar to spawning two threads, one executing each of the two closures. However, the implementation is quite different and incurs very low overhead. The underlying technique is called "work stealing": the Rayon runtime uses a fixed pool of worker threads and attempts to only execute code in parallel when there are idle CPUs to handle it.

When `join` is called from outside the thread pool, the calling thread will block while the closures execute in the pool. When `join` is called within the pool, the calling thread still actively participates in the thread pool. It will begin by executing closure A (on the current thread). While it is doing that, it will advertise closure B as being available for other threads to execute. Once closure A has completed, the current thread will try to execute closure B; if however closure B has been stolen, then it will look for other work while waiting for the thief to fully execute closure B. (This is the typical work-stealing strategy).

Performance：1024*1024 Array random numbers，

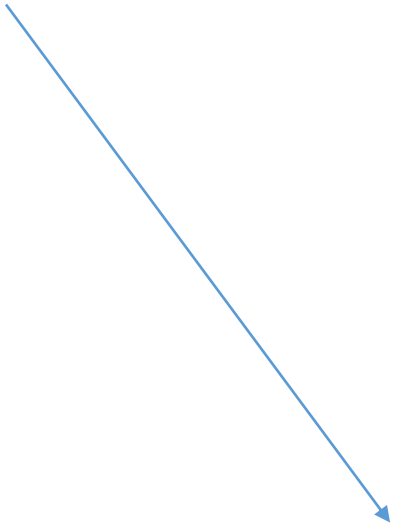
1100 ms （Single thread）

37 ms（Multi-threaded）

19 ms（Multi-threaded using thread pooling and rayon::join）

3. Rayon overhead estimates

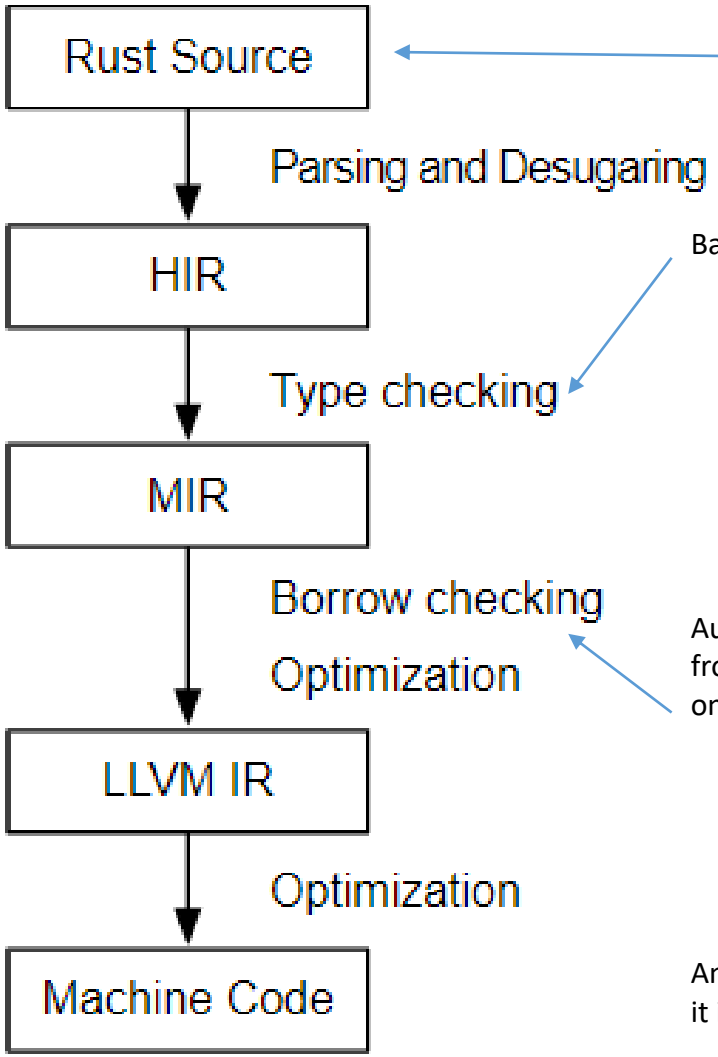
https://github.com/rayon-rs/rayon/blob/fcfd463d503c1f510fc7b47eec636d989c1389f5/src/slice/quicksort.rs#L750



This threshold is necessary when the slice is small.

```
735  /// Sorts `v` recursively.
736  ///
737  /// If the slice had a predecessor in the original array, it is specified as `pred`.
738  ///
739  /// `limit` is the number of allowed imbalanced partitions before switching to `heapsort`. If zero,
740  /// this function will immediately switch to heapsort.
741  fn recurse<'a, T, F>(mut v: &'a mut [T], is_less: &F, mut pred: Option<&'a mut T>, mut limit: u32)
742  where
743      T: Send,
744      F: Fn(&T, &T) -> bool + Sync,
745  {
746      // Slices of up to this length get sorted using insertion sort.
747      const MAX_INSERTION: usize = 20;
748      // If both partitions are up to this length, we continue sequentially. This number is as small
749      // as possible but so that the overhead of Rayon's task scheduling is still negligible.
750      const MAX_SEQUENTIAL: usize = 2000;
751
752      // True if the last partitioning was reasonably balanced.
753      let mut was_balanced = true;
754      // True if the last partitioning didn't shuffle elements (the slice was already partitioned).
755      let mut was_partitioned = true;
756
757      loop {
758          let len = v.len();
759
```

Architectural Design on top of Rust Compiler




```
{
                f(); …
                g();
}
```

Basic solution： From the HIR, using Type Checker to obtain IN/OUT sets of variables from Lambda function descriptions

```
thread::scope(|s|) {
        s.spawn(f);
        s.spawn(g);
}
```

Auto parallelization: From the MIR, using BorrowChecker (Polonius/NLL) to obtain the datalog outputs，
from which we obtain IN/OUT sets of variables at any function call site. For functions f and g, if they are not dependent on each other, we can parallelize them through code transformation on the HIR.

Annotations indicate whether certain variables shall not be included in the IN/OUT sets of variables, and whether it is better not to generate parallel threads.

Extract relevant information from MIR dumps

Using rustc 1.62.0-nightly (52ca603da 2022-04-12)

> rm –rf mir_dump
> rustc –Z dump_mir=y src/main.rs

```
29      println!("Sort numbers in descending order");
30      let mut numbers = [4, 65, 2, -31, 0, 99, 2, 83, 782, 1];
31      println!("Before: {:?}", numbers);
32      quick_sort(&mut numbers, &|x,y| x > y); // IN: {numbers, closure1}, OUT: {numbers}
33      println!("After:  {:?}\n", numbers);
```

Source code

```
用户定义的变量： debug信息暗示
debug numbers => _11;
debug strings => _55;

三个闭包的定义
let mut _153: &[closure@03.rs:32:31: 32:42]; // in scope 1 at 03.rs:32:30: 32:42
let mut _148: &[closure@03.rs:39:31: 39:42]; // in scope 2 at 03.rs:39:30: 39:42
let mut _143: &[closure@03.rs:45:31: 45:54]; // in scope 2 at 03.rs:45:30: 45:54
```

bb5[19]

```
269     bb5: {
270         StorageDead(_13);                // scope 1 at /rustc/52ca603da73
271         StorageDead(_24);                // scope 1 at /rustc/52ca603da73
272         StorageDead(_21);                // scope 1 at /rustc/52ca603da73
273         StorageDead(_20);                // scope 1 at /rustc/52ca603da73
274         StorageDead(_16);                // scope 1 at /rustc/52ca603da73
275         StorageDead(_12);                // scope 1 at /rustc/52ca603da73
276         StorageLive(_25);                // scope 1 at 03.rs:32:5: 32:43
277         StorageLive(_26);                // scope 1 at 03.rs:32:16: 32:28
278         StorageLive(_27);                // scope 1 at 03.rs:32:16: 32:28
279         StorageLive(_28);                // scope 1 at 03.rs:32:16: 32:28
280         _28 = &mut _11;                  // scope 1 at 03.rs:32:16: 32:28
281         _27 = &mut (*_28);               // scope 1 at 03.rs:32:16: 32:28
282         _26 = move _27 as &mut [i32] (Pointer(Unsize)); // scope 1 at 03.
283         StorageDead(_27);                // scope 1 at 03.rs:32:27: 32:28
284         StorageLive(_29);                // scope 1 at 03.rs:32:30: 32:42
285         StorageLive(_30);                // scope 1 at 03.rs:32:30: 32:42
286         _153 = const main::promoted[11]; // scope 1 at 03.rs:32:30: 32:42
287                                          // mir::Constant
288                                          // + span: 03.rs:32:30: 32:42
289                                          // + literal: Const { ty: &[clos
290         _30 = &(*_153);                  // scope 1 at 03.rs:32:30: 32:42
291         _29 = &(*_30);                   // scope 1 at 03.rs:32:30: 32:42
292         _25 = quick_sort::<i32, [closure@03.rs:32:31: 32:42]>(move _26, m
293                                          // mir::Constant
294                                          // + span: 03.rs:32:5: 32:15
295                                          // + literal: Const { ty: for<'r
296     }
```

mir_dump/main.main.002-009.SimplifyCfg-elaborate-drops.after.mir

Compute data dependencies using Borrow Checker Polonius

https://github.com/rust-lang/polonius

The name comes from the famous quote "Neither a borrower nor a lender be", which was said by the character Polonius to his son Laertes in Shakespeare's *Hamlet*.

rustc –Znll-facts src/main.rs

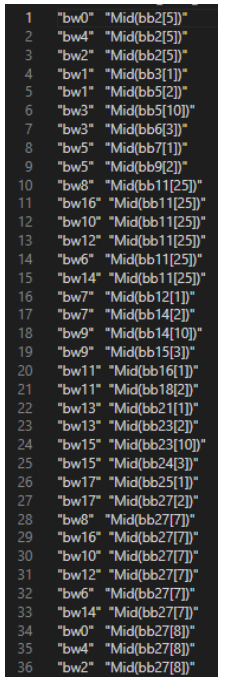
main, main#closure1，2，3, quicksort, partition

- MIR => Datalog facts (Origin, Loan, Point, Variable, Path)
  - loan_issued_at(origin, loan, point)
  - cfg_edge(point, point)
  - loan_killed_at(loan, point)
  - subset_base(origin, origin, point)
  - loan_invalidated_at(point, loan)
  - var_used_at(variable, point)
  - var_defined_at(variable, point)
  - var_dropped_at(variable, point)
  - use_of_var_derefs_origin(variable, origin)
  - drop_of_var_derefs_origin(variable, origin)
  - child_path(path, path)
  - path_is_var(path, variable)
  - path_assigned_at_base(path, point)
  - known_placeholder_subset(origin, origin)
  - Placeholder(origin, loan)

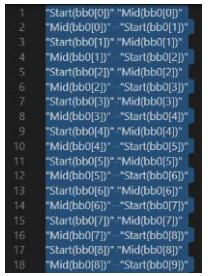
- IN：variable_used_at – loans_invalidated_at
- OUT：variable_defined_at – loans_killed_at

Implementation tools：Polonius + Soufle (Datalog engine)

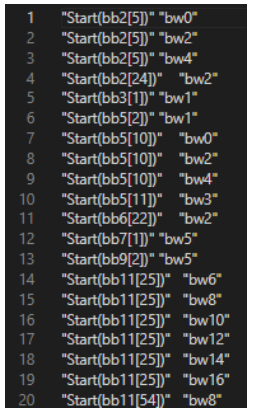
- F_OUT ^ G_IN = {} and F_IN ^ G_OUT = {} => F || G



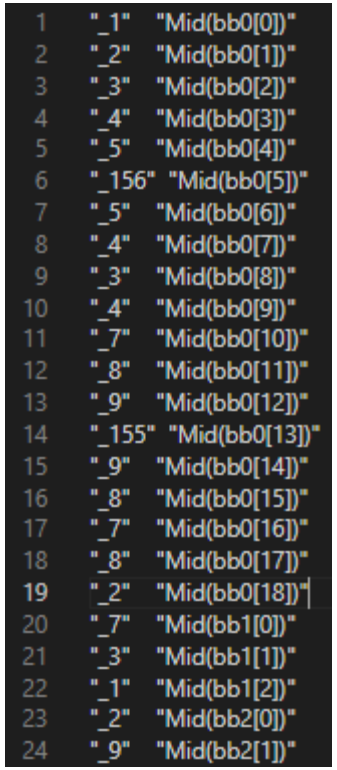
loan_killed_at



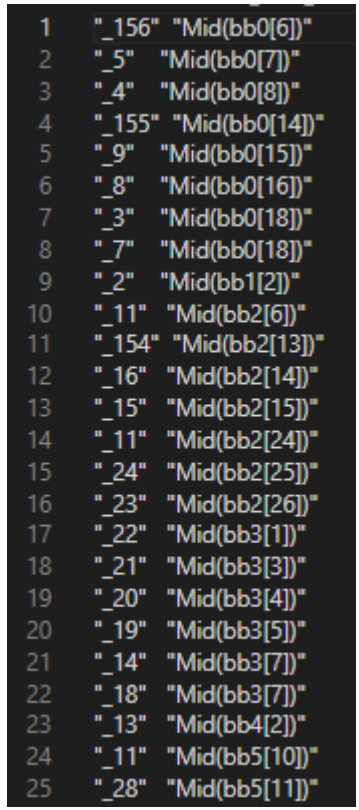
cfg_edge



loan_invalidated_at



var_defined_at



var_used_at

Understanding Polonius facts: we are able to extract IN/OUT data dependencies

> rustc –Z nll-facts src/main.rs

main, main#closure1, main#closure1, main#closure1,main#closure3, quicksort, partition

```
在main/loan_issued_at.facts中:
  "'_#155r" "bw2" "Mid(bb5[10])"    //对应 _28 = &mut _11;
  "'_#156r" "bw3" "Mid(bb5[11])"    //对应 _27 = &mut (*_28);

在main/var_used_at.facts中:
  "_26" "Mid(bb5[19])"
  "_29" "Mid(bb5[19])"

在main/loan_invalidated_at.facts中:
  "Start(bb5[10])"  "bw2" // 在开始先失效，然后重新issue
  "Start(bb5[11])"  "bw3" // The loan bw3 在Start(bb5[11])失效

在main/var_defined_at.facts中:
  "_25" "Mid(bb5[19])"   //_25追溯不到用户变量

在main/loan_killed_at.facts中:
```

```
fraphispre00492 ubuntu rust ~ 2 asciinema play demo.cast
→ ~ cargo new quicksort
    Created binary (application) `quicksort` package
→ ~ cd quicksort
→ quicksort git:(master) x cp -r ../Downloads/demo_dir ./demo_dir
→ quicksort git:(master) x cp demo_dir/demoExample.rs src/main.rs
→ quicksort git:(master) x rm -rf mir_dump
→ quicksort git:( aster) x souffle -F nll-facts/main -D ./IDB demo_dir/compute_INOUT_for_mainCall1.dl
→ quicksort git:(master) x cat IDB/final_in_varNameset.csv
numbers
closure1
→ quicksort git:(master) x cat IDB/final_out_varNameset.csv
numbers
→ quicksort git:(master) x exit
fraphispre00492 ubuntu rust ~
```

Using rustc 1.62.0-nightly (52ca603da 2022-04-12)

> rm –rf mir_dump

> rustc –Z dump_mir=y src/main.rs

```
269     bb5: {
270         StorageDead(_13);                // scope 1 at /rustc/52ca603da73
271         StorageDead(_24);                // scope 1 at /rustc/52ca603da73
272         StorageDead(_21);                // scope 1 at /rustc/52ca603da73
273         StorageDead(_20);                // scope 1 at /rustc/52ca603da73
274         StorageDead(_16);                // scope 1 at /rustc/52ca603da73
275         StorageDead(_12);                // scope 1 at /rustc/52ca603da73
276         StorageLive(_25);                // scope 1 at 03.rs:32:5: 32:43
277         StorageLive(_26);                // scope 1 at 03.rs:32:16: 32:28
278         StorageLive(_27);                // scope 1 at 03.rs:32:16: 32:28
279         StorageLive(_28);                // scope 1 at 03.rs:32:16: 32:28
280         _28 = &mut _11;                  // scope 1 at 03.rs:32:16: 32:28
281         _27 = &mut (*_28);               // scope 1 at 03.rs:32:16: 32:28
282         _26 = move _27 as &mut [i32] (Pointer(Unsize)); // scope 1 at 03.
283         StorageDead(_27);                // scope 1 at 03.rs:32:27: 32:28
284         StorageLive(_29);                // scope 1 at 03.rs:32:30: 32:42
285         StorageLive(_30);                // scope 1 at 03.rs:32:30: 32:42
286         _153 = const main::promoted[11]; // scope 1 at 03.rs:32:30: 32:42
287                                          // mir::Constant
288                                          // + span: 03.rs:32:30: 32:42
289                                          // + literal: Const { ty: &[clos
290         _30 = &(*_153);                  // scope 1 at 03.rs:32:30: 32:42
291         _29 = &(*_30);                   // scope 1 at 03.rs:32:30: 32:42
292         _25 = quick_sort::<i32, [closure@03.rs:32:31: 32:42]>(move _26, m
293                                          // mir::Constant
294                                          // + span: 03.rs:32:5: 32:15
295                                          // + literal: Const { ty: for<'r
296     }
```

bb5[19]

mir_dump/main.main.002-009.SimplifyCfg-elaborate-drops.after.mir

Huawei has contributed over 1/3 of Rust roadmap features since 2020, aspiring our R&D on Rust++

https://trusted-programming.github.io/

Rust community has come up several important roadmaps, where Huawei is making substantial contributions:

- Language Roadmap
- Compiler Team Roadmap
- Library Roadmap

Here is the list of opensource work in progress by Huawei employees. If you want to look at our existing open-source contributions, take a look here.

Compiler

More information on **Diagnostic Translation** here.

More information on **Inline Assembler** here.

More information on **Removing LLVM blockers for Rust compiler** here.

More information on **Polymorphization** here.

More information on **Split DWARF** here.

Language

More information on **FFI Unwind** here.

More information on **Panic Exception handling** here.

Library

More information on **Aspect Oriented Programming (AOP)** here.

More information on **Crypto Library** here.

More information on **ScopedThreads** here.

More information on **parkinglot** here.

More information on **stdarch** here.

Tools

More information on **CRustS** here.

More information on **Clone Detection Tool** here.

More information on **Docs.rs improvements** here.

More information on **Rustup CI improvement** here.

More information on **Rustdoc improvements** here.

We are also initiating the standardization of Rust.

Compared to C/C++ standardization, Rust++ standardization can be seen more as coming up with an invariant list of features that the beloved language should retain during its evolution and innovation.
They offer certainty and guarantee to the adopters of the language.

Next, I will present two examples of such R & D.

1) Adding unsafe behavior classification of assembler;

2) Adding mutex support for fearless concurrency;

RustConf 2021 keynote on Mutex issues

https://www.youtube.com/watch?v=DnYQKWs_7EA

https://github.com/rust-lang/rust/issues/93740

Compared to C pthread_mutex_t, Rust implementation is 36% more performant, and also consuming less 5x less memory per lock

std::sync::RwLock on Linux

Rust 1.61 – pthread_rwlock_t	→	Rust 1.62 – Pure Rust RwLock
Prefers readers: causes writer starvation*		Prefers writers, like Windows SRW locks
Undefined behaviour when moved*		Movable
Undefined behaviour when dropped while locked*		Dropping is a no-op and always safe
Implementation hard to validate/review		Implementation easy to validate/review

std::sync::Mutex on Linux

Speed

	Rust 1.61	Rust 1.62	Speedup
Test 1: extreme contention	58.51 ms	27.65 ms	× 2.1
Test 2: high contention	8.11 ms	1.82 ms	× 4.5
Test 3: low contention (most common case)	2.80 ms	0.78 ms	× 3.6

Total time for 32 threads each locking and unlocking 10'000 times

Tested on a AMD Ryzen Threadripper 3990X 64-Core/128-Thread Processor

std::sync::Mutex on Linux

Size

	Rust 1.61	Rust 1.62
Mutex	16 bytes + 40 bytes allocation	8 bytes + no allocation
RwLock	16 bytes + 48 bytes allocation	8 bytes + no allocation
Condvar	8 bytes + 48 bytes allocation	4 bytes + no allocation

- No more allocator overhead when constructing a Mutex
- No more indirection: better for cache

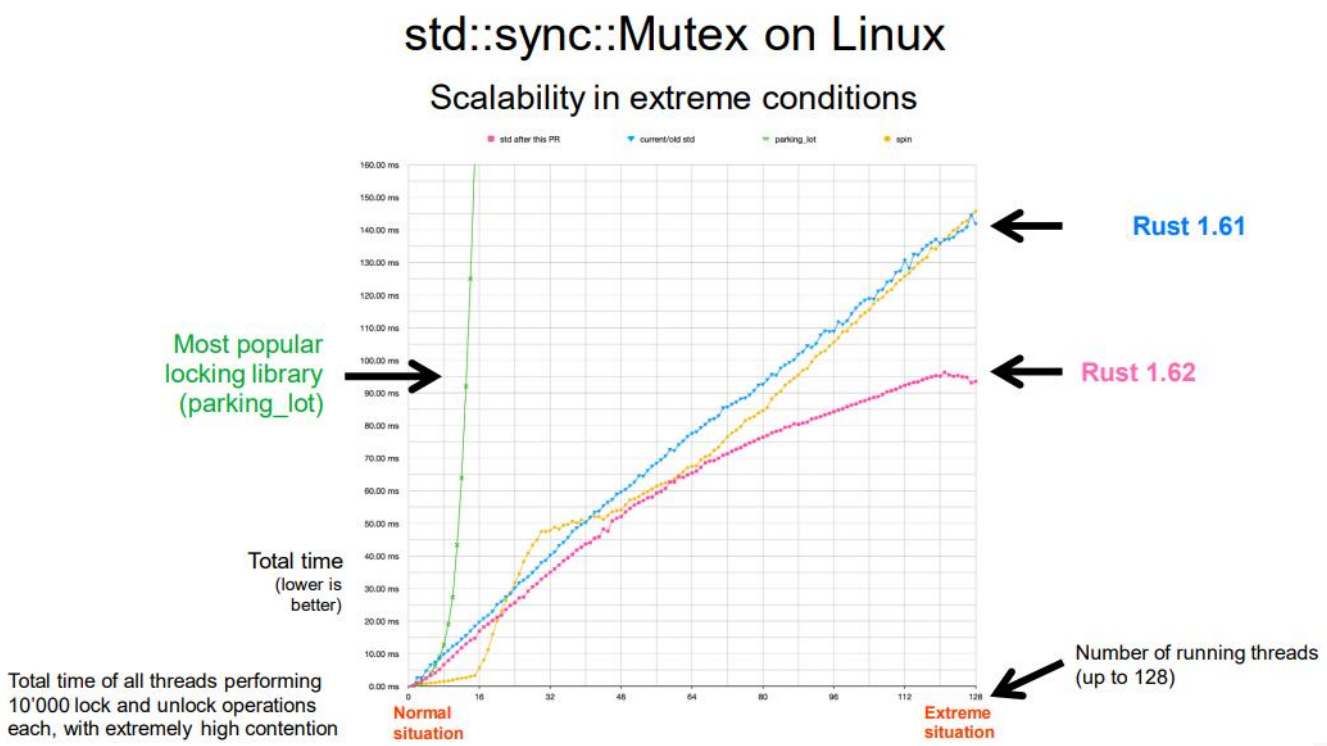
Rust's locks on Linux

std::sync::{Mutex, RwLock, Condvar}

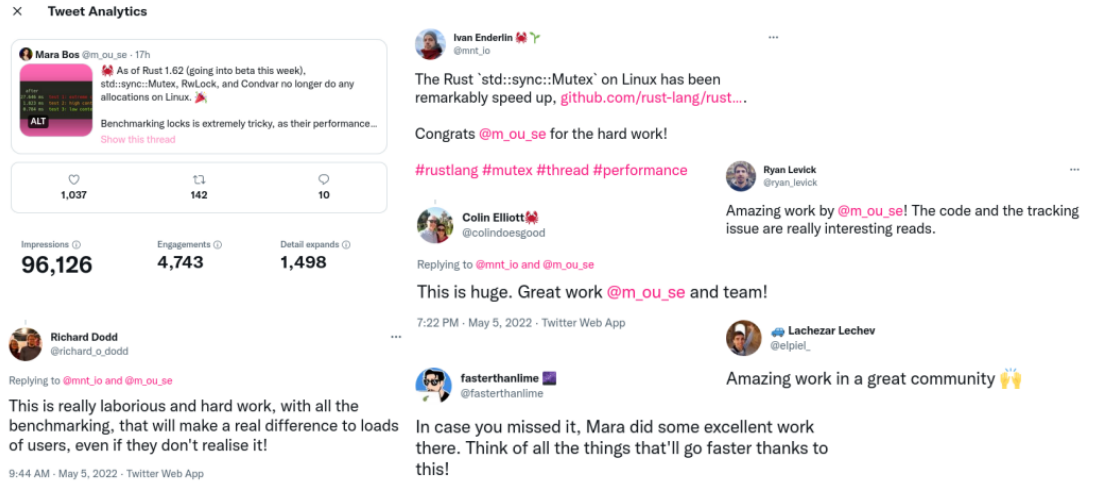
Rust 1.61 – pthread based locks	→	Rust 1.62 – Pure Rust locks
Call to libc/libpthread, dynamically linked		Part of Rust's standard library
Depends on external C library		No external dependencies
Cannot be inlined		Fast path is inlined
Cannot change/improve behaviour, stuck with POSIX and libc ABI		More improvements can be made in Rust, independent from POSIX or C

Algorithm	Safety	Performance	Productivity

It is done for Windows, Linux and BSD, only missing Mac OSX ! Next to explore its application to our products, e.g. Harmony OS

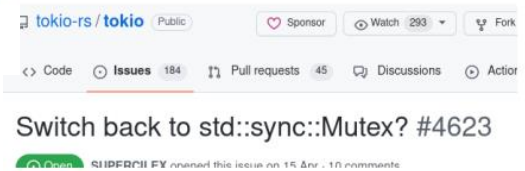


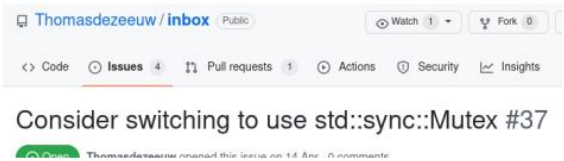
Community reactions

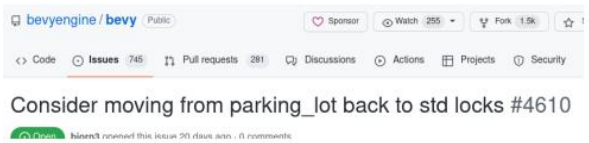


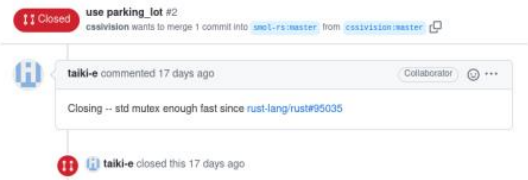
Reactions in open-source projects

Several big open-source projects are discussing the benefits of the new std::sync::Mutex









Can we automatically leverage the performance features using Rust compiler?

Probabilistic Prediction of Undefined Behaviour by Classifying Idiomatic Rust Code and Inline Assembly

```rust
#![feature(asm)]
let x: u64;
unsafe {
    asm!("mov {}, 5", out(reg) x);
}
assert_eq!(x, 5);
```

Figure 2: Inline assembler in Rust through the `asm!` procedural macro, where platform-independent assembler instructions could be embedded along with high-level Rust statements.

Inline Assembly makes Rust on par with C in performance!

```rust
fn safe_read(ptr: &i32) -> i32 { *ptr }
unsafe fn unsafe_read(ptr: *const i32) -> i32 { *ptr }
```

Figure 3: These Rust functions compile to the same assembly.

However, they remove knowledge about unsafe behaviors in Rust code

Observation 1. Rust compiler cannot tell, determinedly, whether a piece of Rust code is `unsafe`.

Observation 2. It is not possible to tell, determinedly, whether a piece of assembly code is `unsafe` through code reviews.

RQ1. Compared to human judgement of undefined behaviour in `unsafe` labels in Rust code, how accurate can machine learning algorithms predict them after training on idiomatic Rust code?

RQ2. Compared to human judgement of undefined behaviour exhibit in the `unsafe` Rust, how accurate can machine learning algorithms predict the assembly implementation without hints from the high-level Rust code constructs?

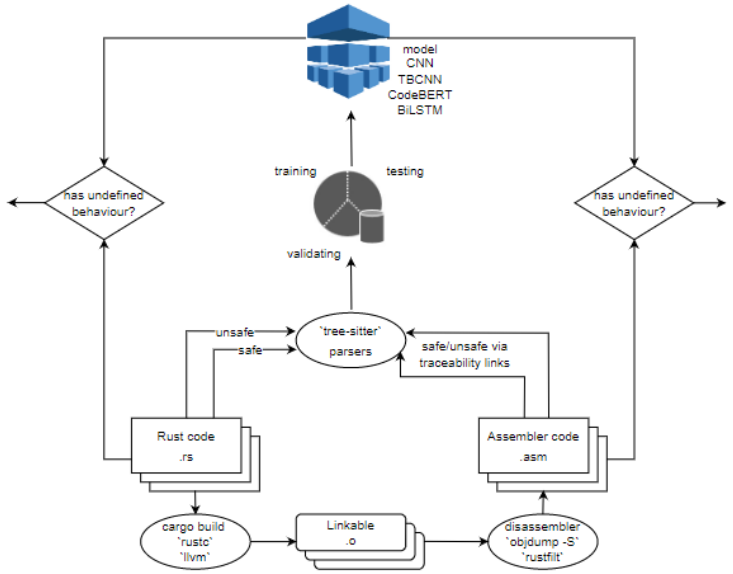


Figure 5: Overview of the predictor architecture

Category	No. Rust functions	No. Asm functions
Safe	42,681	27,100
Unsafe	5,927	3,822

	Safe			Unsafe			Weighted		
Model	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
BiLSTM	0.96	**1.00**	0.98	0.98	0.67	0.79	0.96	0.96	0.96
CNN	0.98	**1.00**	0.99	0.96	0.85	0.90	0.98	0.98	0.98
TBCNN	0.99	0.99	0.99	0.94	0.89	0.91	0.98	0.98	0.98
CodeBERT	**1.00**	**1.00**	**1.00**	**0.98**	**0.96**	**0.97**	**0.99**	**0.99**	**0.99**

```rust
unsafe fn context_backtrace<C>(e: Ref<ErrorImpl>) ->
    Option<&Backtrace>
where
    C: 'static,
{
    let unerased = e.cast::<ErrorImpl<ContextError<C,
        Error>>>().deref();
    let backtrace =
        ErrorImpl::backtrace(unerased._object.error.inner.by_ref());
    Some(backtrace)
}
```

Figure 7: 'Mis-classified' Rust code in 'anyhow-1.0.55/src/error.rs', Lines 810-817

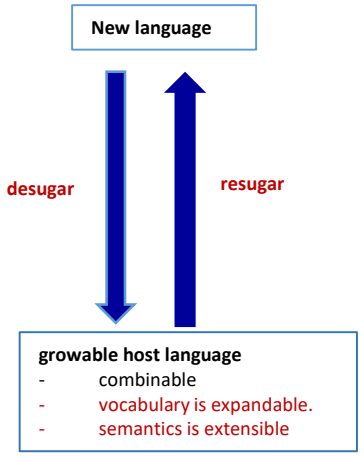
Out perform human experts!

Grow A Functional Programming Language from the Safe Semantics in Rust Core

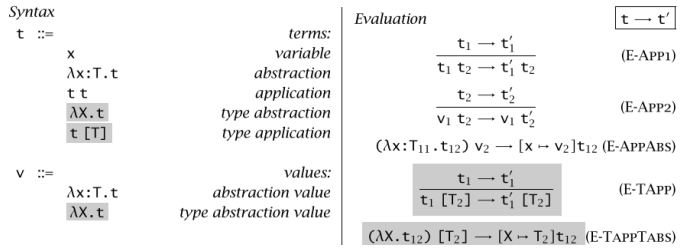
Requirements

The learning curve of Rust code is steep, and C is not safe. Ideally to develop Rust program with C IDE and generate safe C code with Rust code.

Language Definition =
Vocabulary + Grammar + Semantics



Technical scheme: The Rust2C IDE is automatically generated based on the definition of a growable host language based on the syntax sugar Rust/C, and the Rust2C IDE is automatically generated by the bidirectional transformation technology. Lambda calculus System F extends Rust core semantics to support pointers and ownership



RGNVar r t (pointer semantics)
values of type t allocated in region r

RGN rτ(ownership semantics)
computations in region r returning values of typeτ

Automatic generation of an integrated development environment for new languages

New language is defined as follows:
Translate into another existing language
(Syntax Sugar Based on Context-Independent Grammar)

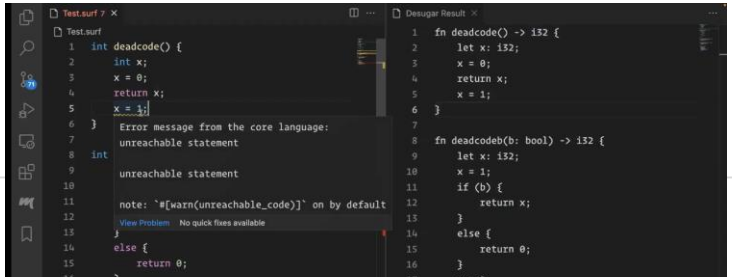
```
struct rectangle {
int x;
int y;
int width;
int height;
};

int arearect(struct rectangle rect) {
return rect.width * rect.height;
}

void main() {
struct rectangle recta;
recta.x = 0;
recta.y = 0;
recta.width = 10;
recta.height = 20;
printf("%d\n", arearect(recta));
}
```

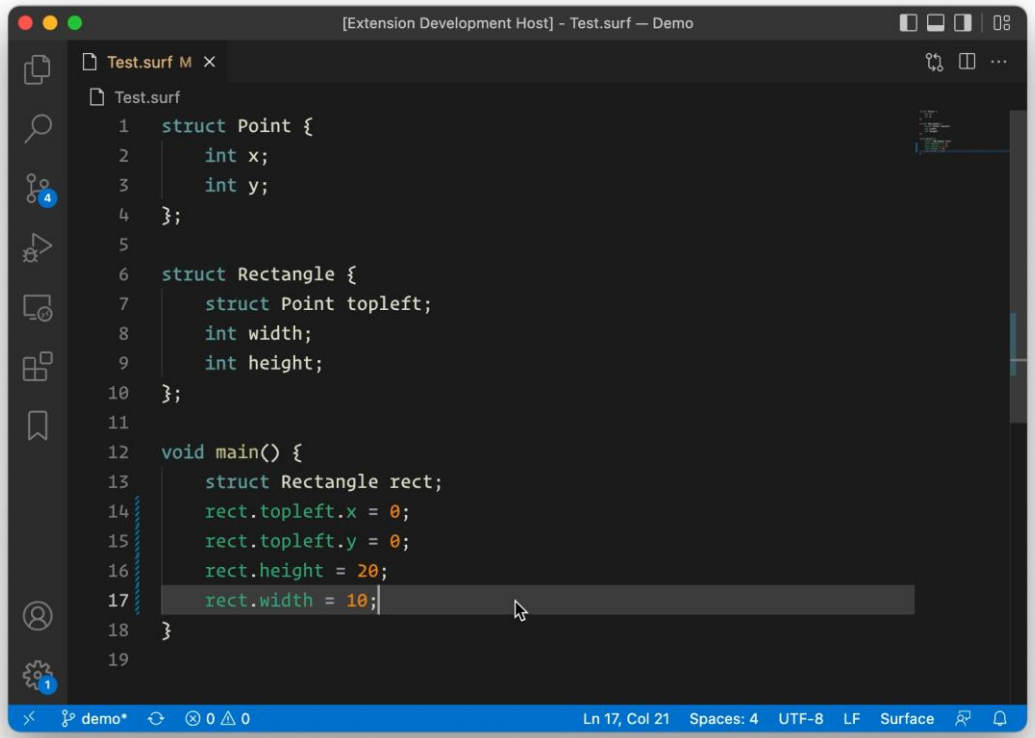
Integrated development environment:
The behavior of the integrated development environment using the Language Server Protocol

C/Rust translation Racket (FP)
surface language internal language



Objectives

- Implements System FR core semantics, supports secure C/Rust languages, and challenges to cover more language features.
- Automatically generate the IDE built based on VSCode. The IDE has the capabilities of code supplementation, type navigation, variable name change, and error checking.



(1) Fluet, Matthew, & Morrisett, Greg. (2004). Monadic regions. Pages 103–114 of: Proceed-ings of the 9th ACM SIGPLAN International Conference on Functional Programming (ICFP'04).
(2) Launchbury, John, & Sabry, Amr. (1997). Monadic state: Axiomatization and type safety. Pages 227–237 of: Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97).

Lower the learning curve of Rust through Machine Learning

Shuofei Zhu, *Ziyi Zhang*, *Boqin Qin*, *Aiping Xiong*, *Linhai Song.* Learning and Programming Challenges of Rust: A Mixed-Methods Study, ICSE 22.

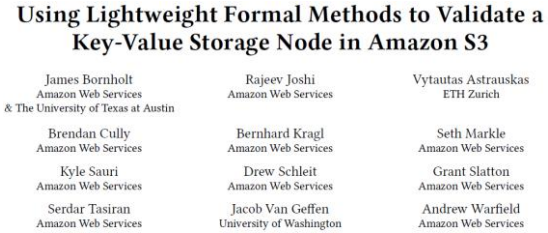
Experience from AWS：
Firecracker, S3
Kani formal verifications

Average 2-4 weeks ← 2-4 months

Machine Learning ← offloading to ← Software Architects Manual Learning


Garbage Collection Makes Rust Easier to Use: A Randomized Controlled Trial of the Bronze Garbage Collector

Michael Coblenz, University of Maryland, College Park, Maryland, USA, mcoblenz@umd.edu
Michelle L. Mazurek, University of Maryland, College Park, Maryland, USA, mmazurek@cs.umd.edu
Michael Hicks, University of Maryland, College Park, Maryland, USA, mwh@cs.umd.edu


Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3

James Bornholt, Amazon Web Services & The University of Texas at Austin
Rajeev Joshi, Amazon Web Services
Vytautas Astrauskas, ETH Zurich
Brendan Cully, Amazon Web Services
Bernhard Kragl, Amazon Web Services
Seth Markle, Amazon Web Services
Kyle Sauri, Amazon Web Services
Drew Schleit, Amazon Web Services
Grant Slatton, Amazon Web Services
Serdar Tasiran, Amazon Web Services
Jacob Van Geffen, University of Washington
Andrew Warfield, Amazon Web Services

GC learning [ICSE22]

unsafe label

lifetime labels

Algorithms learning *AlphaCode from DeepMind*

Rust Simplifications

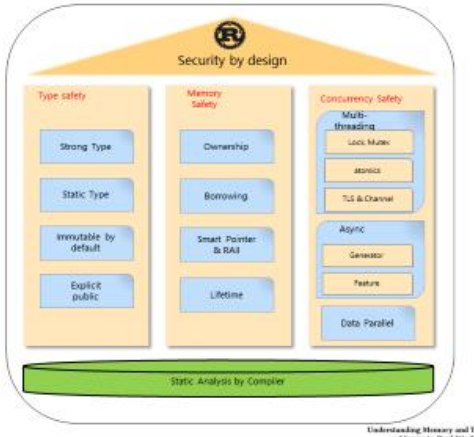
Rust idiomatic code

Rust community experts

Rust code translation tools

Safety

Rust: an efficient language designed to be safe



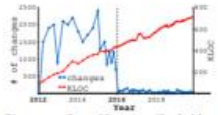


Figure 1. Rust History. *(Each blue point shows the number of feature changes in one release version. Each red point shows total LOC in one release version.)*

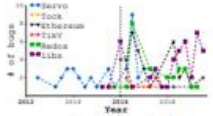


Figure 2. Time of Studied Bugs. *(Each point shows the number of our studied bugs that were patched during a three month period.)*

Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs

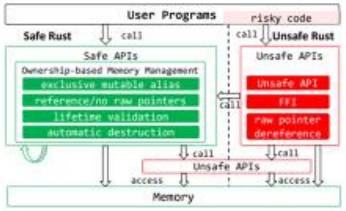


Figure 1: Idea of Rust for preventing memory-safety bugs.

Efficiency

Optimize CRustS to increase the proportion of safe code and the interpretability of safe code

Requirements

To ensure safety, the C code needs to be migrated to the Rust code. Due to the huge legacy of C code in product lines and the steep learning curve of the Rust language, an automated C-to-Rus code conversion tool was developed to assist in manual secure code migration.

Target Product

Product lines with existing C codes and requirements for converting to Rust codes.

Dependencies

The target C code can be compiled normally in the original environment and architecture.

Improving raw pointer to reference refactoring since phase 1

- Enhance the detection capability of time memory safety vulnerabilities for C code that could not be detected by Microsoft Checked C.
- For Rust code that can be translated resulting in a higher safety code ratio:
  - By reducing the number of unsafe regions involving raw pointer expressions

Memory safety issues	CWE Class	No. of Samples	Detected	Ratio
Memory leakage	CWE401	612	234	38.2%
repeated release	CWE415	326	156	46.4%
dangling pointer	CWE416	150	74	49.3%
Heap Buffer Overflow	CWE122	3656	631	17.2%

NIST Juliet 1.3 2017 Dataset C/C++ Memory safety Detection, while Microsoft Checked C could not check

- Michael Ling, Yijun Yu, Haibo Wu, Yuan Wang, James Cordy, Ahmed Hassant. " In Rust we Trust: Translating Unsafe C to Safer Rust", ICSE 2022.
- Emre, Schroeder, Dewey, Hardekopf. "Translating C to Safer Rust", In: OOPSLA, 2021.
- Yamaguchi, Matsuda, David, Wang, "Synsit: Synthesizing Bidirectional Programs using Unidirectional Sketches", in OOPSLA 2021.

A playground has been deployed in Huawei

Code has been released to intersource community

Verify 100% correct and enabled development.

BaseBIOS of Central Hardware

Wireless Q922 Module

Kirin 启动 firmware xLoader 的安全校验模块 Rust 重构项目，该项目是团队内首次使用 Rust 进行产品模块重构的项目，面临人员能力导入困难，项目采用使用 C2Rust 工具对待重构代码进行转译、然后再对转译的 Rust 代码进行优化，极大提升了首次使用 Rust 编码的开发效率；项目组也与 C2Rust 工具团队一起进行了总结，对工具后续优化提供了若干思路，目前该项目已经完成交付，随鸿蒙 3.0 推送手机产品商用。

- Enhance the detection capability of time memory safety vulnerabilities for C code that could not be detected by Microsoft Checked C.
- For Rust code that can be translated resulting in a higher safety code ratio:
  - By reducing the number of unsafe regions involving raw pointer expressions

through the original TXL ceiling, and increases the safe code ratio to 53%, surpassing [OOPSLA21] and our Phase 1 [ICSE22] work (40%). In addition, innovative check analysis algorithms are used to detect time-related memory safety issues. The CWE401, CWE415, CWE416, and CWE122 reach 17% to 49% respectively, surpassing Microsoft Checked C.

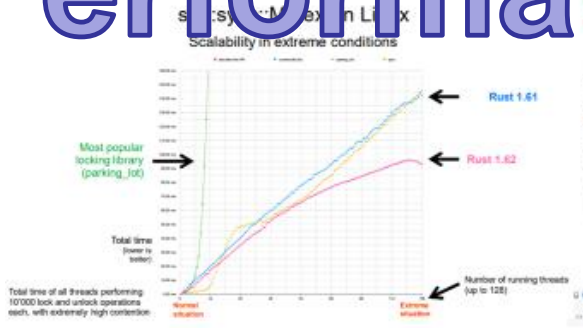
- Develop the Playground demos, which helps the experience and promotion of Rust in the company, and released code and binary to the intersource community to enable C/C++ product line to switch to Rust.
- Wireless Q922 module, BaseBIOS base software pilot and board verification, and HiSilicon Kirin platform sloader: C-to-Rust code conversion functions are complete and correct, which are auxiliary and reference for pilot projects.

HiSilicon Kirin xLoader has been put into commercial use with Harmony 3.0 phones.

Performance

Algorithm | Safety | Performance | Productivity

It is done for Windows, Linux and BSD, only missing Mac OSX ! Next to explore its application to our products, e.g. Harmony OS

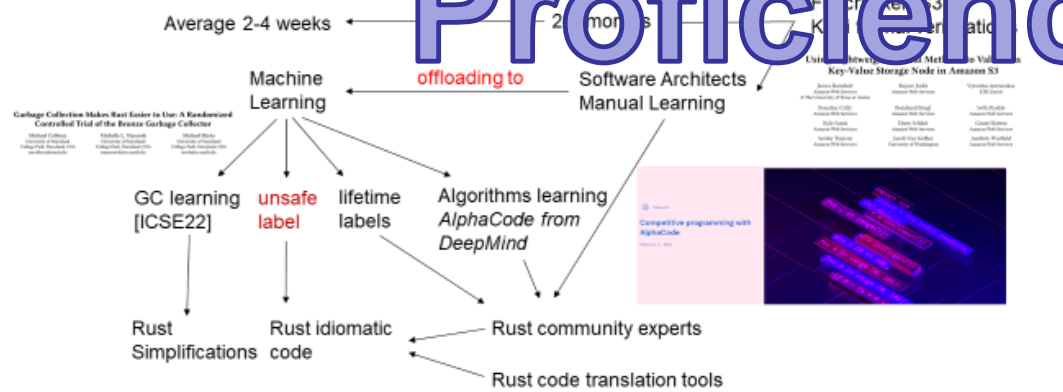


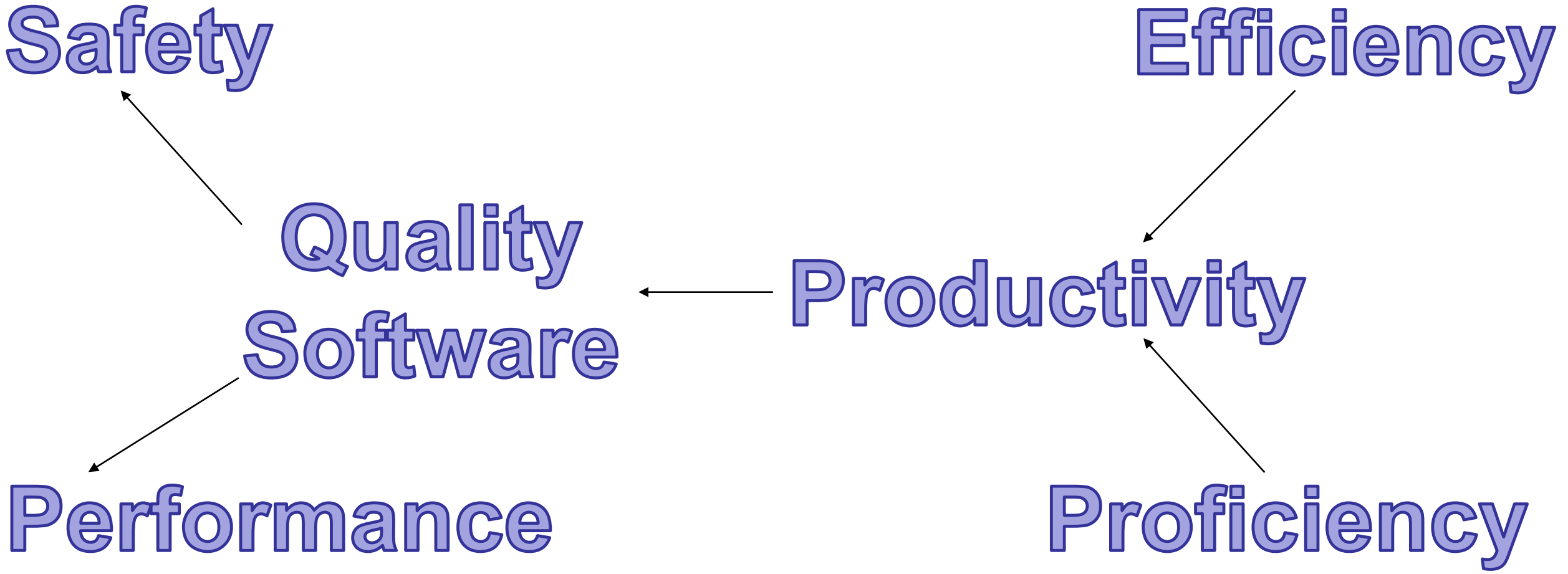
Proficiency

Algorithm | Safety | Performance | Productivity

Lower the learning curve of Rust through Machine Learning

Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, Linhai Song. Learning a... Challenges of Rust: A Mixed-Methods Study, ICSE 22.





Trusted Programming we develop enhance the efficiency and the proficiency of new programming technology towards high productivity in creating high quality software, which guarantee safety and performance.