Trusted Programming: Our Rust Mission at Huawei

Dr. Yijun Yu

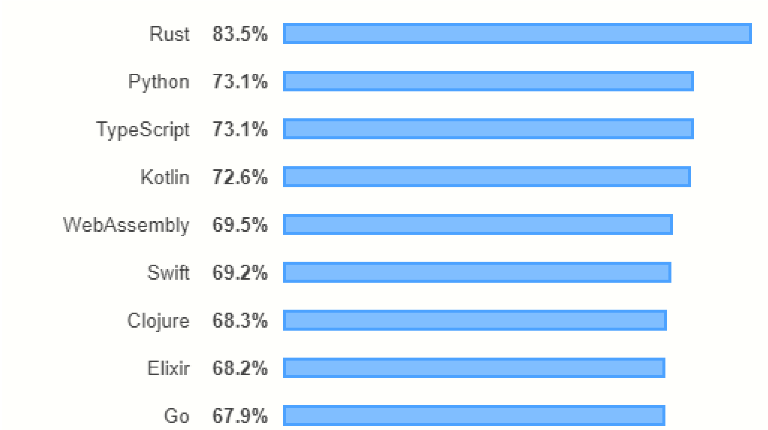
Chief Expert on Trusted Programming
Trustworthy Open-Source Software Engineering Lab
Huawei Technology, Inc.

Dr. Amanieu d'Antras

Principal Rust Expert
Trustworthy Open-Source Software Engineering Lab
Huawei Technology, Inc.

Innovations by Rust

Since 2015, Rust has consistently been voted as the most loved programming language in the StackOverflow survey.



There has also been an increasing number of publications on Rust at recent top programming languages and software engineering conferences.

Title	Creator	Year
The rust language	Matsakis and Klock	2014
Rust for functional programmers	Poss	2014
System Programming in Rust: Beyond Safety	Balasubramanian et al.	2017
Automated refactoring of rust programs	Sam et al.	2017
Verifying Rust Programs with SMACK	Baranowski et al.	2018
K-Rust: An Executable Formal Semantics for Rust	Kan et al.	2018
No Panic! Verification of Rust Programs by Symbolic Execution	Lindner et al.	2018
Leveraging rust types for modular specification and verification	Astrauskas et al.	2019
RustBelt meets relaxed memory	Dang et al.	2019
Stacked borrows: an aliasing model for Rust	Jung et al.	2019
Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software	Yu et al.	2019
Rust编程之道	张汉东	2019
Towards Memory Safe Enclave Programming with Rust-SGX | Proceedings of the 2019 ACM SIGSAC Conferen...		2019
Learning Rust: How Experienced Programmers Leverage Resources to Learn a New Programming Language	Abtahi and Dietz	2020
Is Rust Used Safely by Software Developers?	Evans et al.	2020
Stacked borrows: an aliasing model for Rust	Jung et al.	2020
RustHorn: CHC-Based Verification for Rust Programs	Matsushita et al.	2020
Why scientists are turning to Rust	Perkel	2020
Understanding memory and thread safety practices and issues in real-world Rust programs	Qin et al.	2020
Design of a DSL for Converting Rust Programming Language into RTL	Takano et al.	2020
Memory-Safety Challenge Considered Solved? An Empirical Study with All Rust CVEs	Xu et al.	2020

If that's not enough, a recent Nature 2020 article, 'Why Scientists are Turning to Rust', says that there is increasining momentum on the adoption of Rust amongst scientists.

Why scientists are turning to Rust

Despite having a steep learning curve, the programming language offers speed and safety.

Jeffrey M. Perkel

Initial adoption of Rust at Huawei

At Huawei, we aim to engineer trustworthy software systems in the world's largest telecom industry.

For example, we are working to migrate parts of our code base towards Rust, which is safer and as performant as C/C++. To assist our developers in this process, we are leveraging the open-source C2Rust transpiler to generate Rust code directly from C. We have created automated tools to refactor and clean up this generated Rust code through source-to-source transformations.

We also contribute significant features back to the Rust community. For example, our recent contributions to the Rust compiler enable the compilation of Rust programs for big-endian and ILP32 variants of AArch64. These changes enable Huawei and other hardware companies to run Rust code on networking hardware which commonly use these architecture variants. This contribution is achieved with the help of our Rust expert Amanieu d'Antras, who has pushed through these pull requests to the LLVM compiler, the libc crate, and the Rust compiler itself. These changes introduce new end-to-end cross-compilation targets for the Rust compiler, making it easier to build Rust products for bespoke hardware using a single command:

```
cargo build --target aarch64_be-unknown-linux-gnu
cargo build --target aarch64-unknown-linux-gnu_ilp32
cargo build --target aarch64_be-unknown-linux-gnu_ilp32
```

Finally, we are developing a rich set of internal Rust libraries in Rust built around an actor-based concurrency paradigm. This simplifies asynchronous programming by leveraging Rust language features such as `async`, `await`, etc.

With respect to community engagement, Huawei has been leading the effort in China, hosting the first Chinese Rust Conference during December 26-27 in Shenzhen. We have started to lead the community by carrying out a number of activities, including creating Rust tutorials and Rust coding conventions in Chinese for a vast number of developers who are interested in Rust.

All these factors have led to increased adoption of Rust withing Huawei and a smooth migration from C/C++ programs, which are dominant in the telecom industry. As the leading company in this industry and a founding member of the Rust Foundation, Huawei is committed to the success of Rust and will continue contributing back to the Rust community.

C to Rust transpilation

To illustrate, one can take a close look at the following C program, implementing the '100 doors' algorithm taken from Rosetta Code.

```
#include <stdio.h>

int main()
{
  char is_open[100] = { 0 };
  int pass, door;

  /* do the 100 passes */
```

```
  for (pass = 0; pass < 100; ++pass)
    for (door = pass; door < 100; door += pass+1)
      is_open[door] = !is_open[door];

  /* output the result */
  for (door = 0; door < 100; ++door)
    printf("door #%d is %s.\n", door+1, (is_open[door]? "open" : "closed"));

  return 0;
}
```

The c2rust transpiler initially produces the following Rust code:

```
#![allow(dead_code, mutable_transmutes, non_camel_case_types, non_snake_case,
         non_upper_case_globals, unused_assignments, unused_mut)]
#![register_tool(c2rust)]
#![feature(main, register_tool)]
extern "C" {
    #[no_mangle]
    fn printf(_: *const libc::c_char, _: ...) -> libc::c_int;
}
unsafe fn main_0() -> libc::c_int {
    let mut is_open: [libc::c_char; 100] =
        [0 as libc::c_int as libc::c_char, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
    let mut pass: libc::c_int = 0;
    let mut door: libc::c_int = 0;
    /* do the 100 passes */
    pass = 0 as libc::c_int;
    while pass < 100 as libc::c_int {
        door = pass;
        while door < 100 as libc::c_int {
            is_open[door as usize] =
                (is_open[door as usize] == 0) as libc::c_int as libc::c_char;
            door += pass + 1 as libc::c_int
        }
        pass += 1
    }
    /* output the result */
    door = 0 as libc::c_int;
    while door < 100 as libc::c_int {
        printf(b"door #%d is %s.\n\x00" as *const u8 as *const libc::c_char,
               door + 1 as libc::c_int,
               if is_open[door as usize] as libc::c_int != 0 {
                   b"open\x00" as *const u8 as *const libc::c_char
               } else { b"closed\x00" as *const u8 as *const libc::c_char });
        door += 1
    }
    return 0 as libc::c_int;
}
#[main]
pub fn main() { unsafe { ::std::process::exit(main_0() as i32) } }
```

Although it is compilable, the resulting code is unmaintanable and places too much responsibility on programmers to ensure its safety because all functions are marked as `unsafe`. Although the `c2rust` transpiler supports refactoring commands, it is rather ad hoc, requiring further tweaking with extra help of experienced Rust programmers.

To take advantage of the safety guarantees of the Rust language, we have chosen to leverage TXL, an efficient source-to-source transformation tool. We have developed a robust Rust parser for TXL and used to develop automated refactoring patterns with a guarantee of correct semantics, which allows us to achieve much safer and more maintainable Rust code, e.g.,

```
// #include <stdio.h>
fn main() {
    let mut is_open: [i8; 100] = [0; 100];
    for pass in 0..100 {
        let mut door = pass as usize;
        while door < 100 {
            is_open[door] = !is_open[door];
            door += pass + 1;
        }
    }
    for door in 0..100 {
        print!(
            "door #{} is {}.\n",
            door + 1,
            (if (is_open[door]) != 0 {
                "open"
            } else {
                "closed"
            })
        );
    }
}
```

As one can see, there are no more `unsafe` blocks and the code is fully understandable by programmers.

Adapting end-to-end Rust tooling for Huawei

There are many end-to-end tools out there in the Rust community and we have started to benefit from the interactions with developers of these tools.

Here are just a few examples.

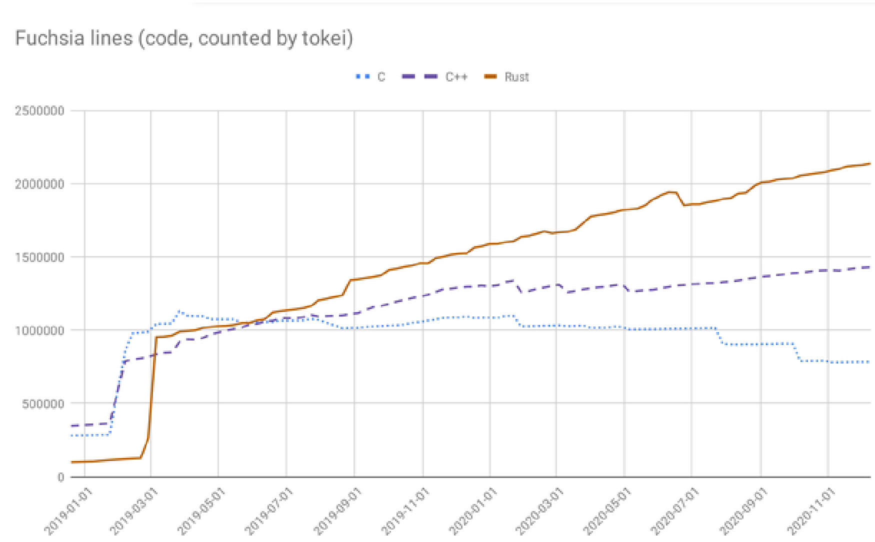
`tokei`

Because trustworthy programming typically involves migrating programming languages, we have adopted `tokei` as our code complexity metric tool, which can recognise as many as 200 languages. For example, the following statistics show

how many lines of code various programming languages have been developed in Google's Fucshia project:

Language	Files	Lines	Code	Comments	Blanks
Assembly	20	33488	29002	60	4426
GNU Style Assembly	215	117405	102142	3763	11500
Autoconf	16	2171	1806	149	216
Automake	1	206	117	36	53
BASH	201	18089	11874	3906	2309
Batch	1	23	20	0	3
C	1742	377591	271884	54594	51113
C Header	6864	832018	508302	185439	138277
CMake	3	285	140	116	29
C++	8162	1860173	1397648	168041	294484
C++ Header	14	10648	10236	204	208
CSS	5	412	341	15	56
Dart	290	37392	27805	5027	4560
Device Tree	8	231	162	40	29
Dockerfile	15	188	158	10	20
Emacs Lisp	1	71	45	12	14
Fish	2	140	84	40	16
FlatBuffers Schema	1	104	80	1	23
GLSL	50	11721	5632	3977	2112
Go	937	172721	137379	14902	20440
Handlebars	30	538	494	4	40
INI	2	18	16	0	2
JavaScript	56	33445	30757	905	1783
JSON	999	68326	68224	0	102
JSX	3	351	299	38	14
LD Script	2	122	108	10	4
Makefile	11	305	195	35	75
Meson	1	12	9	0	3
Module-Definition	5	176	153	0	23
Nix	1	7	6	0	1
Pan	3	41	22	7	12
Perl	41	48582	38835	4941	4806
Pest	5	343	272	35	36
PHP	2	4	3	0	1
Prolog	1	45	34	0	11
Protocol Buffers	22	101560	99748	827	985
Python	318	54621	42039	4652	7930
ReStructuredText	13	2249	1476	0	773
Scala	3	80	67	0	13
Shell	251	28604	19665	5442	3497
SVG	39	6445	6440	2	3
Plain Text	270	129489	0	113931	15558
TOML	445	21574	14792	4089	2693
Vim script	9	419	341	50	28
XML	31	1315	1222	77	16
YAML	262	6474	4933	1134	407

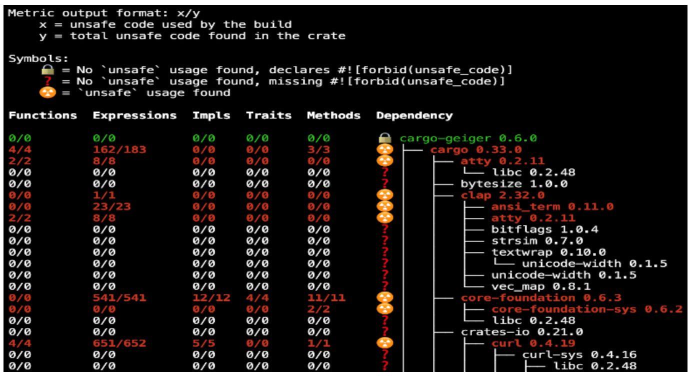
It is relatively easy to plot the proportion of C, C++, Rust code in the evolution of Fucshia, as follows:



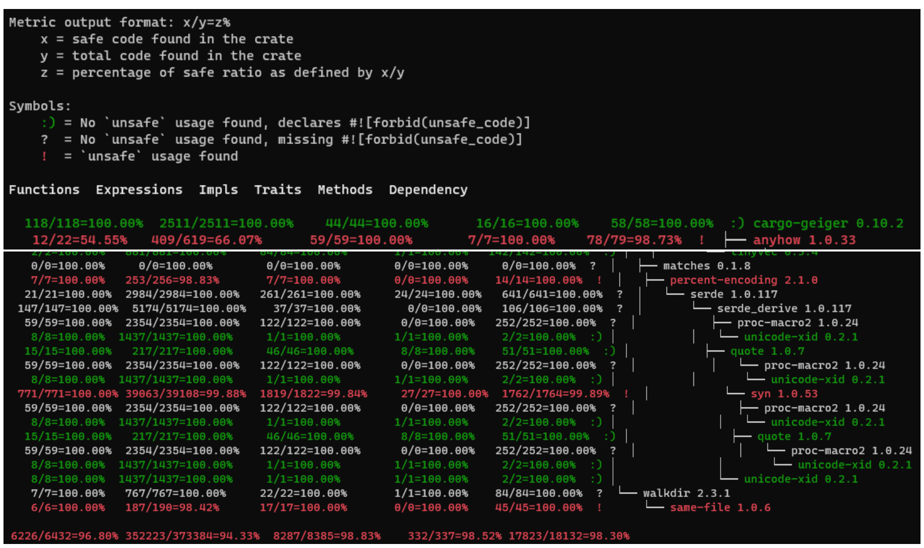
To accommodate the needs to processing multiple programming languages in our projects, we have made a pull request to `tokei` to support batch processing of recognized languages.

`cargo-geiger`

To improve safety, we would like as much code as possible to be checked by the Rust compiler. Fortunately, `cargo-geiger` does almost this by counting the statistics of `unsafe` items such as `fn`, `expr`, `struct`, `impl`, `trait`, and their occurrences in various dependent crates:



However, the statistics do not reflect the ratio of safe items, hence not showing how much has been achieved overall for Rust projects. Therefore, we made a pull request to `cargo-geiger` to report the checked safe ratios of Rust projects. After it was accepted, this tool has been used regularly by our product teams on daily basis. A report will look like the following, which has made it easier to tell which crates have not been fully checked by the Rust compiler:



Research on Rust through Deep Code Learning

As code bases from the Rust open-source community evolve and grow, new developers need to learn the best practices, including but not limited to the language itself. Statistical machine learning methods from large amount of source code, also known as Big Code, has been considered by software engineering research communities: similar to the machine learning problems for image processing and natural language processing where vast number of features requires deep neural networks (DNN) to extract, big code may also be used to train a DNN to reflect on statistical patterns of programs, which is called 'Deep Code Learning'.

In this respect, Huawei is pushing the limits by improving the state-of-the-art of 'cross-language' deep code learning.

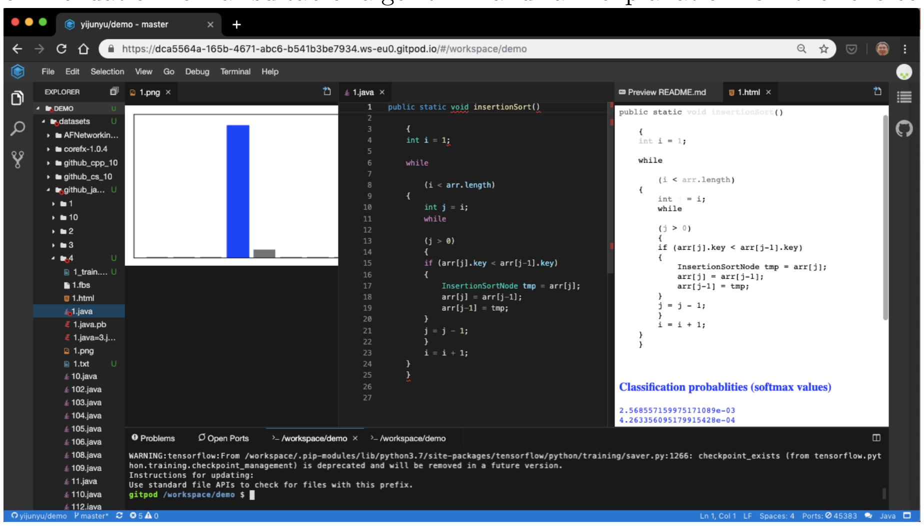
For example, initial deep code learning methods are trained and evaluated using the benchmarks of 52,000 C/C++ programs of 104 algorithm classes collected from the programming courses of Peking University. Traditionally, tree-based convolution neural networks (TBCNN) could achieve 94% accuracy in algorithm classification for this dataset (AAAI'16). A recent progress of the SOTA using abstract syntax trees at the statement level (ICSE'19) achieved 98% accuracy. Our recent progress pushes the SOTA even higher to achieve 98.4% accuracy (AAAI'21) by an innovation on Tree-based Capsule Networks.

Earlier, we have used cross-language datasets to show that the learnt model of one language is applicable to another programming language. For example, using the Rosetta Code datasets from Github, we show it possible to obtain 86% accuracy for algorithm classification (Java to C) (SANER'19), and cross-language API mapping problems (Java to C#) (ESEC/FSE'19). These statistical language models have found multiple applications to software engineering, in terms of code classification, code search, code recommendation, code summary, method name prodiction, and code clone detection (ICSE'21).

To analyse Rust projects, we have made another pull request to the Rust parser project `tree-sitter` and XML serialization crate `quick-xml`, which allow us to feed the abstract syntax trees of Rust programs to train a deep code learning model. The preliminary results are quite promising, the detection algorithms in Rust can reach an accuracy as high as 85.5%. This number is still climbing as we continue working on improving toolchains.

Conclusion

In summary, the Huawei Trustworthy Open-Source Software Engineering Lab is working hard to provide programmers an end-to-end IDE toolchain that intelligently assists in maximizing safety and performance. A prototype of such an IDE is shown as an extension to the Visual Studio Code where programmers are assisted with the recommendation of a suitable algorithm and an explanation of the choice.



A journey towards the vision of Trusted Programming has just begun and we hope to work collaboratively with the Rust community, and the upcoming Rust Foundation, to lead a smooth revolution to the Telecom software industry.