4 Code Organization

Thomas Prokosch
November 6, 2018
Outline

1. Functions
2. Paths
3. Modules
4. Crates
5. Testing and benchmarking
6. Writing documentation
Functions
Nested functions

Rust supports **nested functions**. This is especially useful with recursion where it is often desirable to **hide the recursive helper** function:

```rust
fn mscg(&self, s: &Subst) -> (Subst, Subst, Subst) {
    fn mscg_rec(e: &u64, e: &Expr, s: &Subst,
        mut m: Subst, mut s1: Subst, mut s2: Subst)
        -> (Subst, Subst, Subst) {
        // ... some computation...
    }
    let mut (m, s1, s2) = (Subst::new(), Subst::new(), Subst::new());
    for (i, e) in &self.map {
        let (m_, s1_, s2_) = mscg_rec(i, e, s, m, s1, s2);
        m = m_; s1 = s1_; s2 = s2_;
    }
    (m, s1, s2)
}
```

Thomas Prokosch
Paths
What are paths?

Paths are a sequence of path components separated by the namespace qualifier ::

x; // single path component
x::y::z; // accessing z via path x::y
"42".parse::<i32>(); // "turbofish": type arg to generic fn
Keywords `self`, `super`

- Paths starting with `::` are **global paths** relative to the **crate root**, see later
- `self` refer to the same module
- `super` refer to the **parent** module (can be repeatedly used)

`self`, `super` keywords can also be used for calling functions.
use Declarations

- A use declaration creates a **new name binding** for a path.
- The `as` modifier allows to define a **new local name**.
- The asterisk `*` wildcard matches **all paths** with a given prefix.

**Prelude**

- is a module
- **loaded by default** by Rust
- contains commonly used types (such as `std::option::Option<T>`), functions, and macros (e.g. `println!`)

```rust
use std::prelude;
```

implici{tly inserted into every Rust program which automatically brings commonly used functions into scope
Nested use declarations

- Braces and commas allow to **group paths** with a common prefix.
- The nested use syntax may also be used with the as modifier.
- Nesting may also occur **recursively**.

For example,
```
use a::b::{self, c, d::e as baz, f::* , g::{h, i}}; resolves to
use a::b;
use a::b::c;
use a::b::d::e as baz;
use a::b::f::*;
use a::b::g::h;
use a::b::g::i;
```
Modules
What is a module?

- **organizes code** by partitioning it, possibly **nested**
- introduces a **namespace** and privacy
- part of a **crate** ("library", discussed later)
- introduced with keyword **mod**

```rust
fn main() { phrases::greetings::hello(); } 
mod phrases {
    fn hello() {
        println!("Hello, world!"); 
    }
    pub mod greetings {
        pub fn hello() {
            super::hello();
        }
    }
}
```

Thomas Prokosch
By default, everything in Rust is private:

- **private** items can be used from **current or child modules**
- keyword **pub** makes items **public**:
  - functions: `pub fn`
  - modules: `pub mod`
  - structs, enums: `pub struct, pub enum`
  - etc.

**What does "public" mean?**

Items can be accessed from within a module $m$ if items from module $m$ can be accessed.
Re-exporting with `pub use`

**Module hierarchy not necessarily appropriate** as a public interface

Solution:
- Create new module to be used as public interface, re-exporting types and functions.
- Advantage: **Commonly used functions** brought into scope with a **single** `use` statement.
- Less commonly used functions still require individual `use` statements or the full path.

Example: Rust prelude, or other large crates

```rust
pub use self::implementation::api;
// allows the use of self::api::f()
mod implementation {
    pub mod api {
        pub fn f() {}  // }
    }
}
```

Thomas Prokosch
Module separation using files

- `mod creates namespace`, no physical separation
- better: every module in its own file
- **file name is module name**
- **syntax**: `mod IDENT;` loads module `IDENT` from file `IDENT.rs`

```rust
// file: main.rs
mod greetings; // import "greetings.rs"
fn main() { greetings::hello(); }

// file: greetings.rs
// no mod declaration: file itself is the module
pub fn hello() { println!("Hello, world!"); }
```
Module separation using directories

- We know: File **m.rs** defines a module named "m".
- Alternative: In the current directory there exists a **directory m** which contains a **file mod.rs**.
- This directory may itself contain files/directories as sub-modules.

**Attention:** Rust is about to **change some semantics** in this area.
Crates
What is a crate?

A crate

- is a **compilation unit**
- contains an implicit, un-named top-level module
- produces either a
  - **binary** from src/main.rs, or a
  - **library** from src/lib.rs

**Crate dependencies**

**Dependency** to a crate must be reflected in

1. respective source files (before `use`) with `extern crate CRATENAME;`
2. Cargo.toml file
Cargo as a package manager

Cargo

- fetches and **builds dependencies** of a project
- runs **Rust compiler** `rustc`, integrates with other compilation steps for example, a C compiler such as GCC
- runs **tests and benchmarks**

**The cargo build subcommand**

<table>
<thead>
<tr>
<th>build profile</th>
<th>compilation</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev (default)</td>
<td>fast</td>
<td>slow binaries with debug symbols</td>
</tr>
<tr>
<td>release</td>
<td>slow</td>
<td>fast optimized binaries</td>
</tr>
</tbody>
</table>

See invocations to the `rustc` rust compiler by running cargo verbosely.

Thomas Prokosch
Cargo.toml and Cargo.lock

Cargo.toml contains **hand-specified** dependencies

Cargo.lock **exact version information** based on Cargo.toml to get a **reproducible** build environment

- **executable projects**: put Cargo.lock in repository
- **library projects**: do not include Cargo.lock in repository

**Remark**: With this infrastructure in place, Rust is well-positioned to support **reproducible research**. To quote Wikipedia:

*The term reproducible research refers to the idea that the ultimate product of academic research is the paper along with the laboratory notebooks [...] and full computational environment used to produce the results in the paper [...]*

Without reproducibility, it is difficult/impossible to base **new research on previous results**, even within the same research group.

Thomas Prokosch
Specifying dependencies in Cargo.toml

Dependencies are specified beneath a [dependencies] section in Cargo.toml:

<table>
<thead>
<tr>
<th>dependency on</th>
<th>syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>crates.io</td>
<td><code>$DEP = &quot;\$VERSION&quot;</code></td>
</tr>
<tr>
<td>local path</td>
<td><code>$DEP = { path = &quot;\$PATH&quot; }</code></td>
</tr>
<tr>
<td>latest master commit</td>
<td><code>$DEP = { git = &quot;\$URL&quot; }</code></td>
</tr>
<tr>
<td>latest branch commit</td>
<td><code>$DEP = { git = &quot;\$URL&quot;, branch = &quot;$BR&quot; }</code></td>
</tr>
<tr>
<td>given git tag</td>
<td><code>$DEP = { git = &quot;$URL&quot;, tag = &quot;$TAG&quot; }</code></td>
</tr>
<tr>
<td>specific git revision</td>
<td><code>$DEP = { git = &quot;$URL&quot;, rev = &quot;$HASH&quot; }</code></td>
</tr>
</tbody>
</table>

Exact version is recorded in Cargo.lock until dependency update is requested:

```
cargo update -h
```
Linking C with a build script

Linking with C/C++ requires a `build.rs` build script in the project root directory and additions to `Cargo.toml`:

```
[package]
# ...
links = "foo"       # for linking with libfoo.a
build = "build.rs"
```

The `build script` `build.rs` should perform the following tasks:

1. build the library
2. find the library
3. select static/dynamic linking by writing proper output
4. expose C headers

For simpler libraries, the `cc crate` can be used.
crates.io is the authoritative server for third-party libraries.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cargo login</td>
<td>store API token for further use</td>
</tr>
<tr>
<td>cargo package</td>
<td>create crate file, in target/package/*.crate</td>
</tr>
<tr>
<td>cargo publish</td>
<td>upload the crate</td>
</tr>
</tbody>
</table>

Support for alternate crate servers is being worked on.
Testing and benchmarking
attributes `#[test], #[cfg(test)]`

- `#[test]` attribute on a function indicates that the function is a test
- `#[cfg(test)]` **compiles conditionally**: only if tests are run — to be enabled with `cargo test`

Good practice to associate this attribute with a module (function visibility!)

```rust
#[cfg(test)] // only compiles when running tests
mod tests {
    use super::greet; // import root greet function

    #[test]
    fn test_greet() {
        assert_eq!("Hello, world!", greet());
    }
}
```

Thomas Prokosch
In tests, values need to be **compared to default values.**

<table>
<thead>
<tr>
<th>macro</th>
<th>ensures</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>panic!</td>
<td>-</td>
<td>if a+b != 30 { panic!() }</td>
</tr>
<tr>
<td>assert!</td>
<td>truth</td>
<td>assert!(a+b==30, &quot;a={}, b={}&quot;, a, b);</td>
</tr>
<tr>
<td>assert_eq!</td>
<td>equality</td>
<td>assert_eq!(4, 2+2);</td>
</tr>
<tr>
<td>assert_ne!</td>
<td>difference</td>
<td>assert_ne!(1, 0);</td>
</tr>
</tbody>
</table>
Running tests

cargo test will run the following tests:

- **unit tests** marked with `#[cfg(test)]` (files from the library/binary)
- **integration tests** contained in `tests/*.rs` files (no sub-directories considered)
- **documentation tests** contained in documentation blocks

This ensures that documentation is up-to-date with the current code base.

The result of a test run is the number of successful tests, and the names of the failing ones.

**Benchmarks**
work similar to tests but are only available in Rust nightly.
QuickCheck technique

Inventing test cases can be tedious. QuickCheck by Koen Claessen and John Hughes (2000)

- is a combinator library
- generates test cases
- checks test cases against specification
- If a test case is found that does not conform to specification, test case is shrunk to find minimal test case.

Some implementation work for custom data types.

Implemented by these crates:

- proptest
- quickcheck
Writing documentation
Unlike C++ but like Haskell Rust offers to **generate documentation** from source code.

- Open Rust documentation with `rustup doc`.
- Build package documentation with `cargo doc`.
- Read package documentation with `cargo doc --open`.

Thomas Prokosch
## Comments

<table>
<thead>
<tr>
<th>comments</th>
<th>line</th>
<th>block</th>
</tr>
</thead>
<tbody>
<tr>
<td>outer doc (describes following)</td>
<td>/// doc</td>
<td>/** doc */</td>
</tr>
<tr>
<td>inner doc (describes parent)</td>
<td>///! doc</td>
<td>*/! doc */</td>
</tr>
</tbody>
</table>

Convention: Avoid block comments.
Documentation

- uses Markdown syntax of unspecified flavor
- with **code blocks** (three backticks ‘ ‘ ‘ as opening and closing delimiters) allows to include **examples**
- examples are **tested** automatically when tests are run

```rust
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let five = 5;
/// ```
///
/// assert_eq!(6, my_crate::add_one(5));
/// ```
```

```rust
tpub fn add_one(x: i32) -> i32 { x + 1 }
```
Today’s goals:

1. Be able to test and document code.
2. Know how to write and use modules and crates.

We have learned about…

- paths and use declarations
- the difference between modules and crates
- restricting visibility of data types and functions
- integration of other code into the Rust build system
- testing code (with assert! and QuickCheck)
- writing documentation