Nightly Rust

CIS 198 Lecture 15

Nightly Rust

- AKA all those cool features you aren't allowed to use on the homework.
- Multirust:
  - `multirust update nightly`: download or update the nightly toolchain
  - `multirust override nightly`
  - `multirust override nightly-04-01`
- With the standard rustup script:
  - `curl [rustup.sh] | sh -s -- --channel=nightly`

Feature Gates

- Unstable or experimental features are behind feature gates.

```
#![feature(plugin_registrar, rustc_private)]
```

- Include the appropriate attributes at the top of your root module.

Compiler Plugins

- Add custom behavior during compilation process.
- A few different categories that run at different times during compilation:
    - Syntax extension/procedural macro: code generation.
    - Lint pass: code verification (e.g. style checks, common errors).
    - LLVM pass: transformation or optimization of generated LLVM.

Compiler Plugins

- To use a plugin in your crate:

```
#![feature(plugin)]
#![plugin(foo)]
#![plugin(foo(bar, baz))]
```

- If arguments are present, rustc passes them to the plugin.
- Don't use `extern crate ...`; this would link against the entire crate, including linking against all of its dependencies.

Compiler Plugins

- To write a plugin:

```
// Unlock feature gates first.
#![feature(plugin_registrar, rustc_private)]

fn expand_hex_literals(cx: &mut ExtCtxt, sp: Span,
          args: &[TokenTree]) -> Box<MacResult + 'static> {
    /* Define the "hex" macro here. */
}


// Register your macro here.
#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("hex", expand_hex_literals);
}
```

Procedural Macros

- Similar to "normal" macros; call them with the same `foo!` pattern.
- Typical macros generate code by matching against syntax patterns.
- Compiler plugins run Rust code to manipulate the syntax tree directly.
- Benefits:
  - More powerful code generation.
  - Compile-time input validation.
  - Custom error messages (reported on original source code, not generated code)

Procedural Macros

- Example: Docopt, a command line argument parser plugin.

```
docopt!(Args derive Debug, "
Naval Fate.

Usage:
  naval_fate.py ship new <name>...
  naval_fate.py ship shoot <x> <y>

Options:
  -v --verbose  Verbose output.
  --speed=<kn>  Speed in knots [default: 10].
");
```

- Verify at compile time if you have a valid configuration.
- Generates the **Args** struct, which **derives `Debug`**.
  - At runtime, this contains information about the flags provided.

Procedural Macros

- Very recent RFC (PR) proposing a major change to the macro system.
  - (Literally, the PR for this RFC was opened since I wrote last week's lecture and said there might be a new macro system in the future. -Kai)
- Decoupling macros from compiler's internal `libsyntax`, using `libmacro` instead.

```
#[macro]
pub fn foo(TokenStream, &mut MacroContext) -> TokenStream;
```

Syntex-syntax

- A pre-processing workaround for compiler plugins for stable Rust.
  - Part of `serde`, used by many other projects.
- Uses Cargo support for `build.rs`: a pre-compilation "script".
  - Compiles and runs before the rest of your crate
- `syntex` uses an external build of `libsyntax` to run any code generation that would be part of a compiler plugin.

Lints

- These plugins are run after code generation.
- Lints traverse the syntax tree and examine particular nodes.
- Throw errors or warnings when they come across certain patterns.
- As simple as `snake_case`, or more complicated patterns, e.g.:

```
if (x) {
    return true;
} else {
    return false;
}
```

- `rust-clippy` is a collection of common mistakes.

LLVM Passes

- LLVM is an intermediate stage between source code and assembly language.
- LLVM-pass plugins let you manipulate the LLVM-generation step in compilation
- Actual plugin must be written separately in C++.
  - Need to register with plugin registrar so it can be called at the appropriate time.
- Examples: extremely low-level optimization, benchmarking.

Inline Assembly

- For the particularly daring, you can insert assembly directly into your programs!
- Both feature gated *and* unsafe.
- Specify target architectures, with fallbacks.
- Directly binds to LLVM's inline assembler expressions.

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn foo() {
    unsafe {
        asm!("NOP");
    }
}

// other platforms
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }
```

No stdlib

- Another crate-level attribute: `#![no_std]`.
- Drops most of standard libary: threads, networking, heap allocation, I/O, etc.
- What's left is the `core` crate:
  - No upstream or system libraries (including libc).
  - Primitive types, marker traits (`Clone`, `Copy`, `Send`, `Sync`), operators.
  - Low-level memory operations (`memcmp`, `memcpy`, `memswap`).
  - `Option`, `Result`.
- Useful for building operating systems (Redox), embedded systems.

No stdlib

- `#![no_std]` also drops `fn main()`.
- Annotate your starting function wtih `#[start]`.
  - Same function signature as main() in C.

```
// Entry point for this program
#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    0
}
```

Specialization

- Generally, a trait or type may only have one `impl`.
  - Multiple `impl`s with different trait bounds are allowed, if they don't conflict.
- Trait implementations for generic types may therefore restrictively constrain operations on those types.
- Specialization enables you to provide multiple `impl`s for a type, so long as one is clearly *more specific* than another.

Specialization

```
trait AddAssign<Rhs=Self> {
    fn add_assign(&mut self, Rhs);
}

impl<R, T: Add<R> + Clone> AddAssign for T {
    fn add_assign(&mut self, rhs: R) {
        let tmp = self.clone() + rhs;
        *self = tmp;
    }
}
```

Specialization

```
trait Extend<A> {
    fn extend<T>(&mut self, iterable: T)
        where T: IntoIterator<Item=A>;
}
```

- Collections that implement the `Extend` trait can insert data from arbitrary iterators.
- The definition above means that nothing can be assumed about `T` except that it can be turned into an iterator.
  - All code must insert elements one at a time!
- E.g. extending a `Vec` with a slice could be done more efficiently, and the compiler can't always figure this out.

Specialization

- `Extend` could be specialized for a `Vec` and a slice like so.
    - `std` doesn't do this *yet*, but it's planned to do so soon.

```rust
// General implementation (note the `default fn`)
impl<A, T> Extend<A, T> for Vec<A> where T: IntoIterator<Item=A> {
    default fn extend(&mut self, iterable: T) {
        // Push each element into the `Vec` one at a time
    }
}

// Specialized implementation
impl<'a, A> Extend<A, &'a [A]> for Vec<A> {
    fn extend(&mut self, iterable: &'a [A]) {
        // Use ptr::write to write the slice to the `Vec` en masse
    }
}
```

Specialization

- Specialization also allows for *default specialized* trait implementations:

```
trait Add<Rhs=Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
    fn add_assign(&mut self, rhs: Rhs);;;;
}

default impl<T: Clone, Rhs> Add<Rhs> for T {
    fn add_assign(&mut self, rhs: Rhs) {
        let tmp = self.clone() + rhs;
        *self = tmp;
    }
}
```

Specialization

- Currently in Rust, trait implementations may not overlap.

```
trait Example {
    type Output;
    fn generate(self) -> Self::Output;
}

impl<T> Example for T {
    type Output = Box<T>;
    fn generate(self) -> Box<T> { Box::new(self) }
}

impl Example for bool {
    type Output = bool;
    fn generate(self) -> bool { self }
}
```

Specialization

- Specialization is designed to allow implementation overlap in specific ways.
  - `impl`s may overlap in concrete type or type constraints.
- One `impl` must be more specific in some way than others.
  - This may mean different concrete types (`T` vs. `String`)...
  - ...or different type bounds (`T` vs. `T: Clone`)

Specialization

- Remember, with any of these trait implementations all dispatch is strictly static unless you're using trait objects.
- Trait specialization should strictly improve the performance of your code!
  - Assuming you actually write everything correctly, of course.
- Available on the Rust 1.9 Nightly: `#![feature(specialization)]`

`const` Functions

- Motivation: types like `UnsafeCell` are more unsafe than they need to be.

```
struct UnsafeCell<T> { pub value: T }
struct AtomicUsize { v: UnsafeCell<usize> }
const ATOMIC_USIZE_INIT: AtomicUsize = AtomicUsize {
    v: UnsafeCell { value: 0 }
};
```

`const` Functions

- Proposed solution: `const fn`.
  - Kind of similar to `constexpr` from C++.
- Certain functions and methods may be marked as `const`, indicating that they may be evaluated at compile time.
- `const fn` expressions are pretty restrictive right now.
  - Arguments must be taken by value
  - No side effects are allowed (assignment, non-`const` function calls, etc.)
  - No instantiating types that implement `Drop`
  - No branching (if/else, loops)
  - No virtual dispatch