IN RUST USING MACROS

METAPROGRAMMING

METAPROGRAMMING IN RUST USING MACROS

Generics

Declarative Macros

Procedural Macros

Compiler Plugins

DECLARATIVE MACROS

- Declare starting with `macro_rules!`
  - macro_rules! bar {<rules>}
- Call with a postfixed `!`
  - bar!(<args>)
- Fewer syntactic restrictions on macro arguments compared to Rust code
- Emits Rust code in-place

SYNTAX PATTERN MATCHING

- Rules defined in macros use pattern matching on syntax
- (pattern) => { <emitted code> };
    - Simple example:

```
macro_rules! foo {
    (x) => (println!("It is x!“));
    (y) => (println!("It is y!"));
}
```

META-VARIABLES

- Extracted through syntax patterns, starting with a `$`
- Requires a fragment specifier to define its token type
  - E.g.: ident, path, expr, ty
- Simple example:

```
macro_rules! bar {
    (x => $e:expr) => (println!("mode X: {}", $e));
    (y => $e:expr) => (println!("mode Y: {}", $e));
}
```

REPETITION

- Base pattern: $($x:expr)*
  - 0 or more expressions, separated by spaces
- Repetition specifiers besides `*`:
  - +: One or more repetitions
  - ?: Zero or one occurrences
- Optionally use separators before the repetition specifiers to require separation of repeated arguments (e.g. `$($x:expr),*` for separation by comma)

MACRO HYGIENE

- What about name collisions on expansion?
- Macro expansion happens in a distinct ‘syntax context’!
- Side Effect:
    - Variable names created in a macro are not available in calling context
    - Solution: pass desired variable as $ident type to the macro

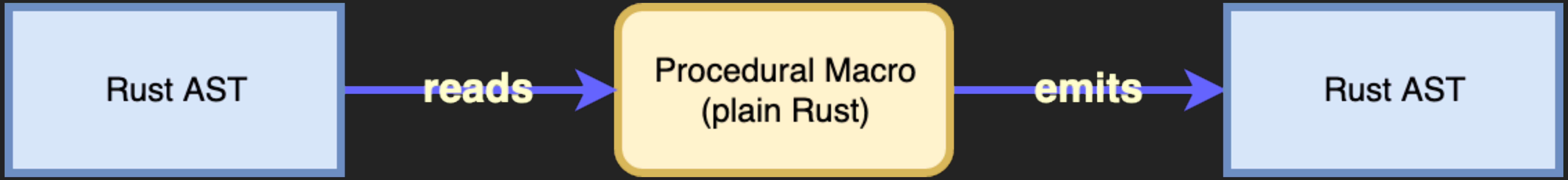
$crate

- Special Meta Variable to allow macros to call functions from its own crate
- Expands either to „::<crate_name>“
  - Where <crate_name> is the name of the macro’s crate if included outside of the crate
- …or expands to nothing if it is used inside the macro’s crate

EXPORT AND IMPORT

- #[macro_export] macro_rules! …
  - The macro is visible outside the macro's crate
  - All other macros are private to the macro's crate
- #[macro_use] extern crate …
  - Obsolete!
  - Since Rust 1.30 macros can simply be imported with „use" statements

PROCEDURAL MACROS



```
#[proc_macro]
pub fn my_macro(input: TokenStream) -> TokenStream {…}
```

PROCEDURAL MACROS

- Written in plain Rust
- Rust standard libraries and crates available
- Assembling Token Streams manually is tedious
  - Use the „quote“ macro!
  - Converts Rust code into a Token Stream, including Rust variable capture

PROCEDURAL MACROS

- Derive Macros
  - To implement Traits for Structs and Enums (e.g.: Default, Copy, Debug)
- Attribute-like Macros
  - Define custom attributes for Structs, Enums and Functions
- Function-like Macros
  - AST passed as function argument

PROCEDURAL MACROS APPLICATIONS

- Generating Rust Trait implementations
- Transforming foreign syntax (e.g. SQL) to Rust code
- Generating Rust code from interface description files
- Generating interface description files from Rust code
- Ad-hoc DSLs

DISCUSSION