Traits and lifetimes in Rust

Cseh Viktor – Bolyai presentation

Rust primer

- C-like language
- Has similar structs, operators, datastructures, etc.
- Bundled build system called `cargo`
- Extensive type-safe macro system

Enums in Rust

- Corresponds to std::variant in C++
- Pattern matching is a first-class citizen
- For “non-null” types, the compiler will optimize the layout

Traits in Rust

- No inheritance between structs in Rust
- Interfaces, called `trait`s
- New traits can be implemented for existing objects
- It can contain functions, associated types, and constants

Traits and inheritance

- Supertraits – one trait must implement another
- Auto traits – implemented on all objects by default, “markers”
- Traits cannot have conflicting implementations
    - No “overload resolution” in Rust

References and lifetimes

- Any object in Rust can have a reference to it
  - In C++ references are just non-null const pointers
  - In Rust, references have lifetime information
  - Mutable reference cannot exist when there are immutable references
- Pointers do not have lifetime guarantees, dereferencing is unsafe

Lifetime analysis

- Lifetimes are checked statically at compile-time
- “borrowck”: All references live until the end of the scope
  - Introduces “useless” scopes
- “Non-lexical lifetimes”: References can end before a scope
  - Flow-insensitive – rejects control-flow-dependent correct programs
- “Polonius”: Future lifetime checker using data-flow analysis
  - Will be flow-sensitive

Lifetime relations

- “Outlives” relation between lifetimes
- A reference must outlive the container it’s stored in
  - Generic parameter to the struct
  - Same generic parameters can be applied to traits as well
- This makes linked lists difficult to implement
  - Each node must outlive the nodes it refers to
- Self-referential structs cannot be implemented
  - Polonius will enable this

Moving objects in Rust

- In C++, move semantics guarantee that an object stays in place if required
- In Rust, core assumption is that all objects can be moved freely
    - In some cases, that is unwanted behavior – Pin<T>
- Clones are explicit, but “Copy”-able types are rare