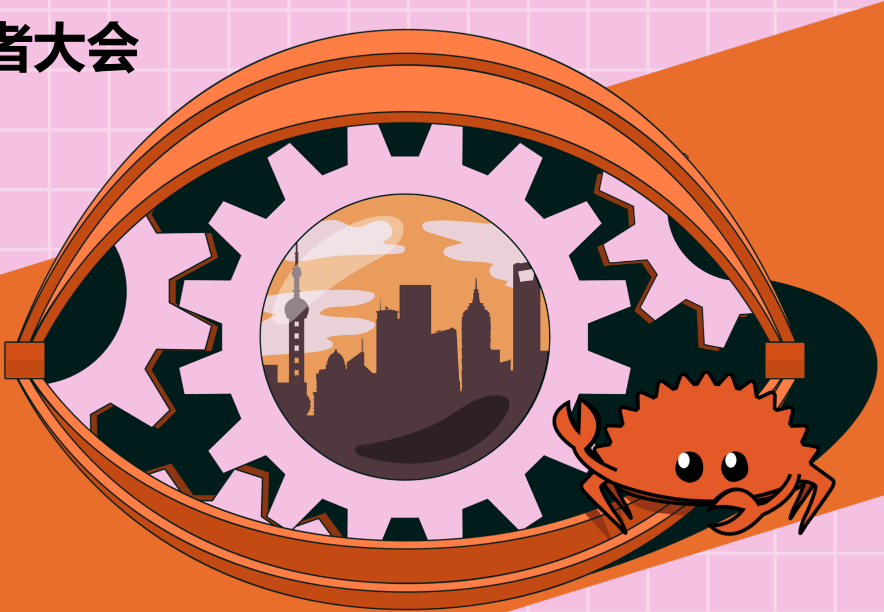
RUST CHINA CONF 2023

第三届中国Rust开发者大会



6.17-6.18 @Shanghai

Rust China Conf 2023

Candid: the interface description language of the Internet Computer smart contracts

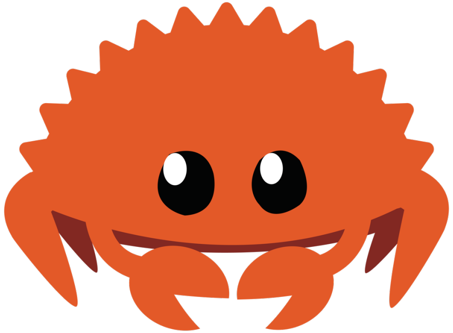
-- How DFINITY uses procedure macro to extend Rust

Yan Chen

DFINITY Foundation

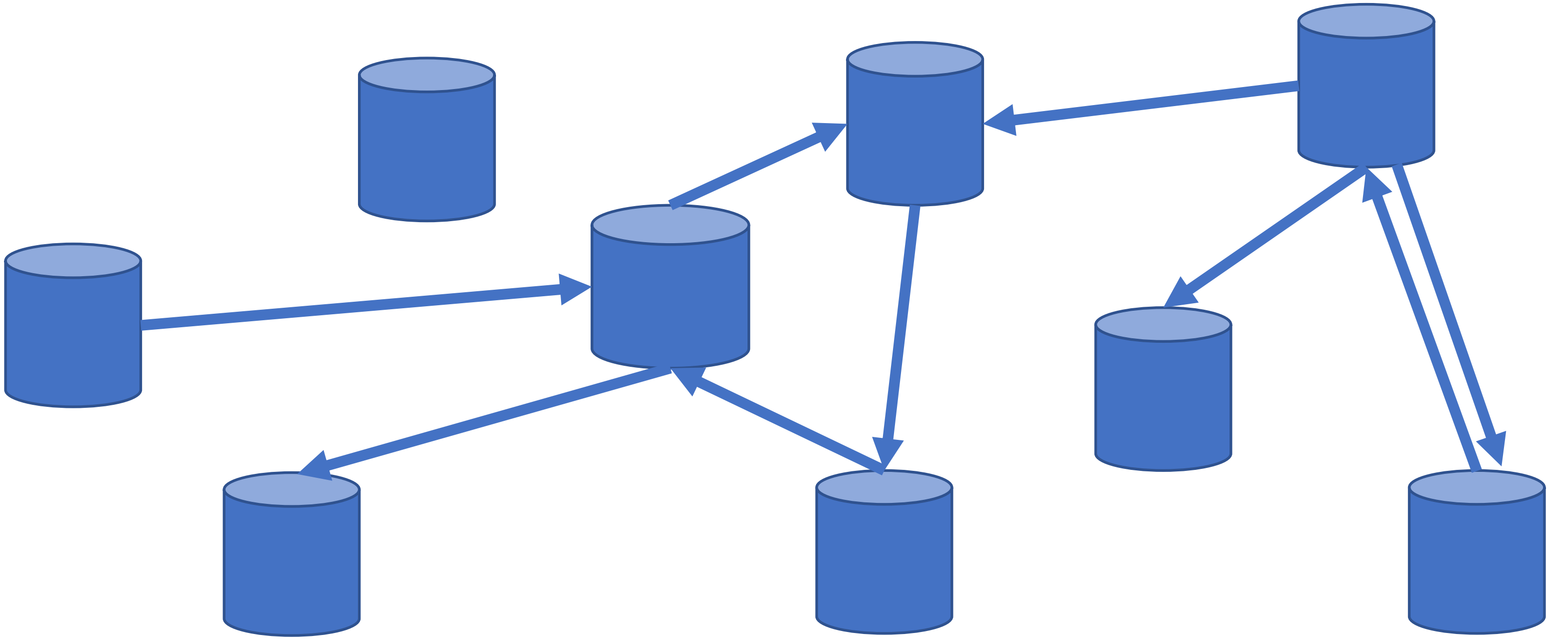
Overview

- Macro system is a hidden gem in Rust.
- We share our experience in safely extending Rust languages without modifying the compiler.
  - Running example: Candid, a strongly typed serialization library
  - Extended language features:
    - Backward compatible API upgrades with subtyping
    - Type reflection
    - Structural typing



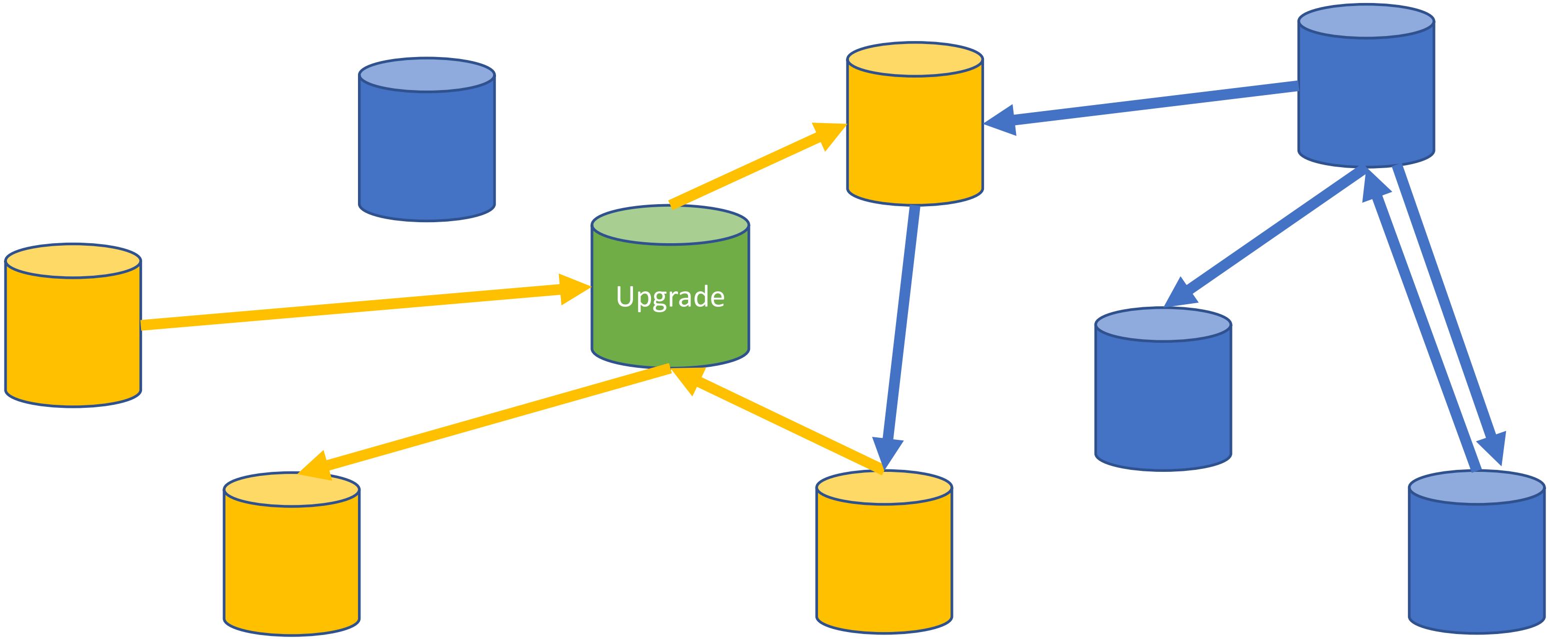
Motivation

- How to maintain interoperability between microservices?



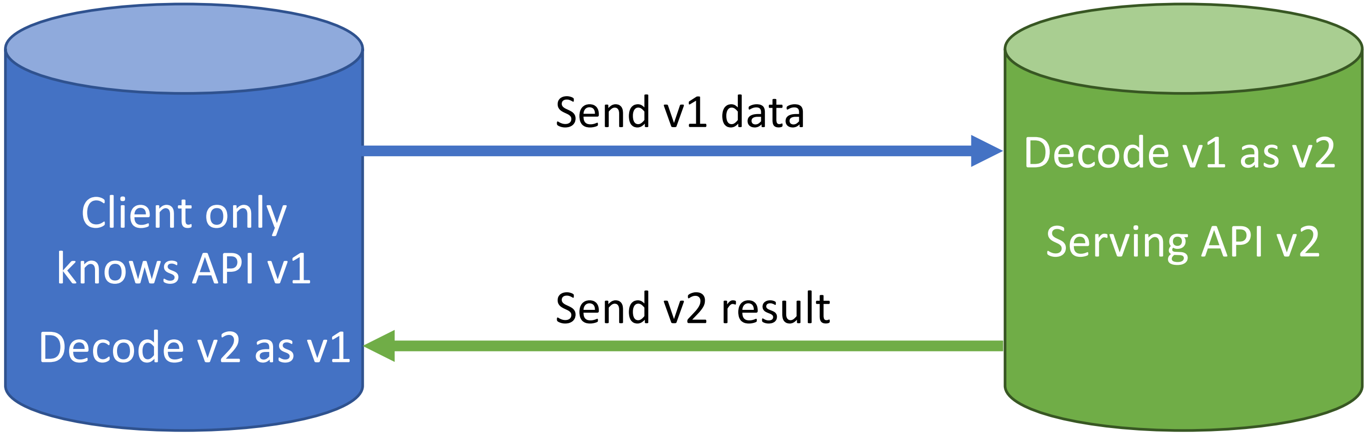
Motivation

- How to maintain interoperability between microservices?



Motivation

- How to maintain interoperability between microservices?
  - APIs are allowed to evolve without breaking existing client
    - Wire format contains type information
    - Decoder knows two types: wire type, and expected type



Client only knows API v1

Decode v2 as v1

Send v1 data

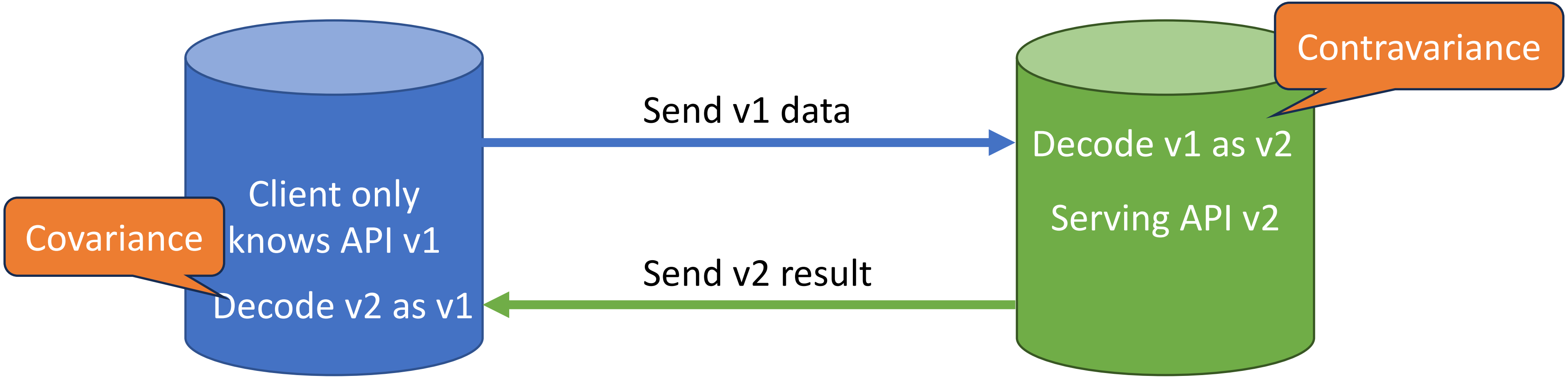
Decode v1 as v2

Serving API v2

Send v2 result

Motivation

- How to maintain interoperability between microservices?
  - APIs are allowed to evolve without breaking existing client
    - Wire format contains type information
    - Decoder knows two types: wire type, and expected type



Client only knows API v1

Decode v2 as v1

Covariance

Send v1 data

Send v2 result

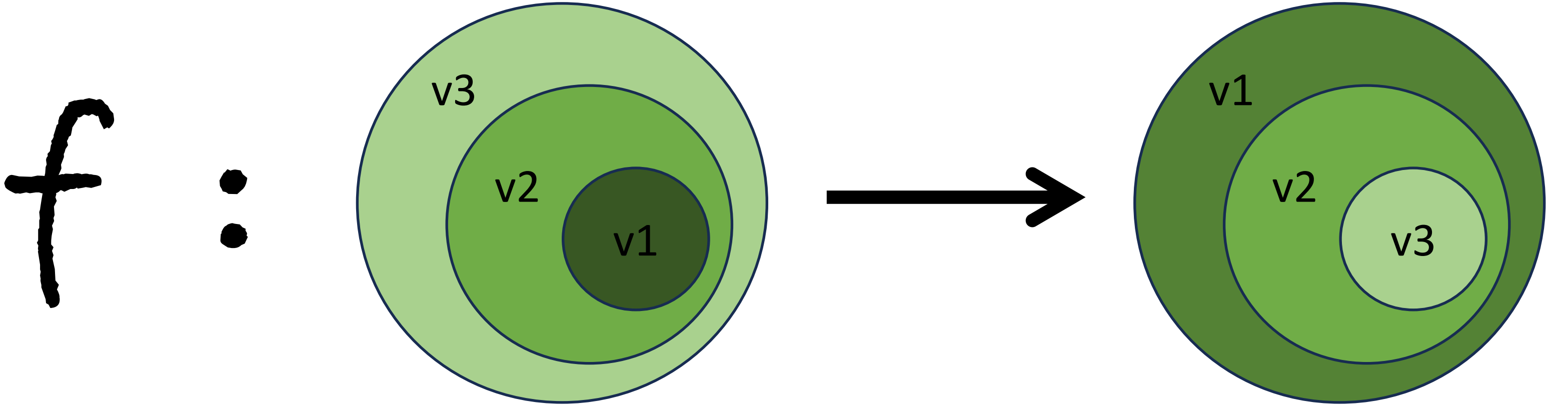
Decode v1 as v2

Serving API v2

Contravariance

Subtyping and safe upgrades

- Outbound data (provided by service) can upgrade to more specific data
- Inbound data (required by service) can upgrade to more general data
- Orientation is inverted for higher order functions



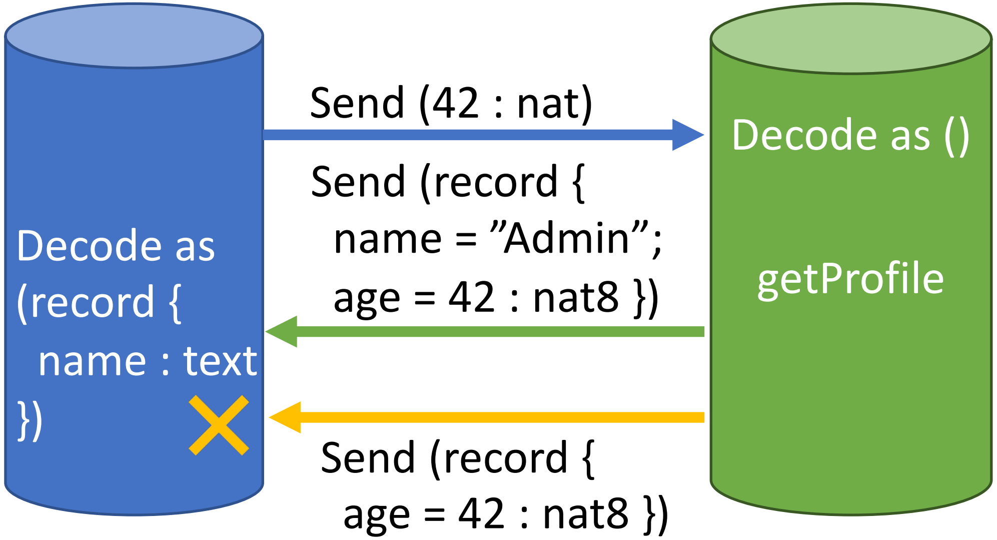
Example

API v1

```
type Profile = record { name : text };
service : {
  getProfile : (nat) -> (Profile);
}
```

API v2

```
type Profile = record {
  name : text; age : nat8
};
service : {
  getProfile : (~~nat~~) -> (Profile);
}
```



Decode as (record {
  name : text
})

Send (42 : nat)

Send (record {
  name = ”Admin”;
  age = 42 : nat8 })

Send (record {
  age = 42 : nat8 })

Decode as ()

getProfile

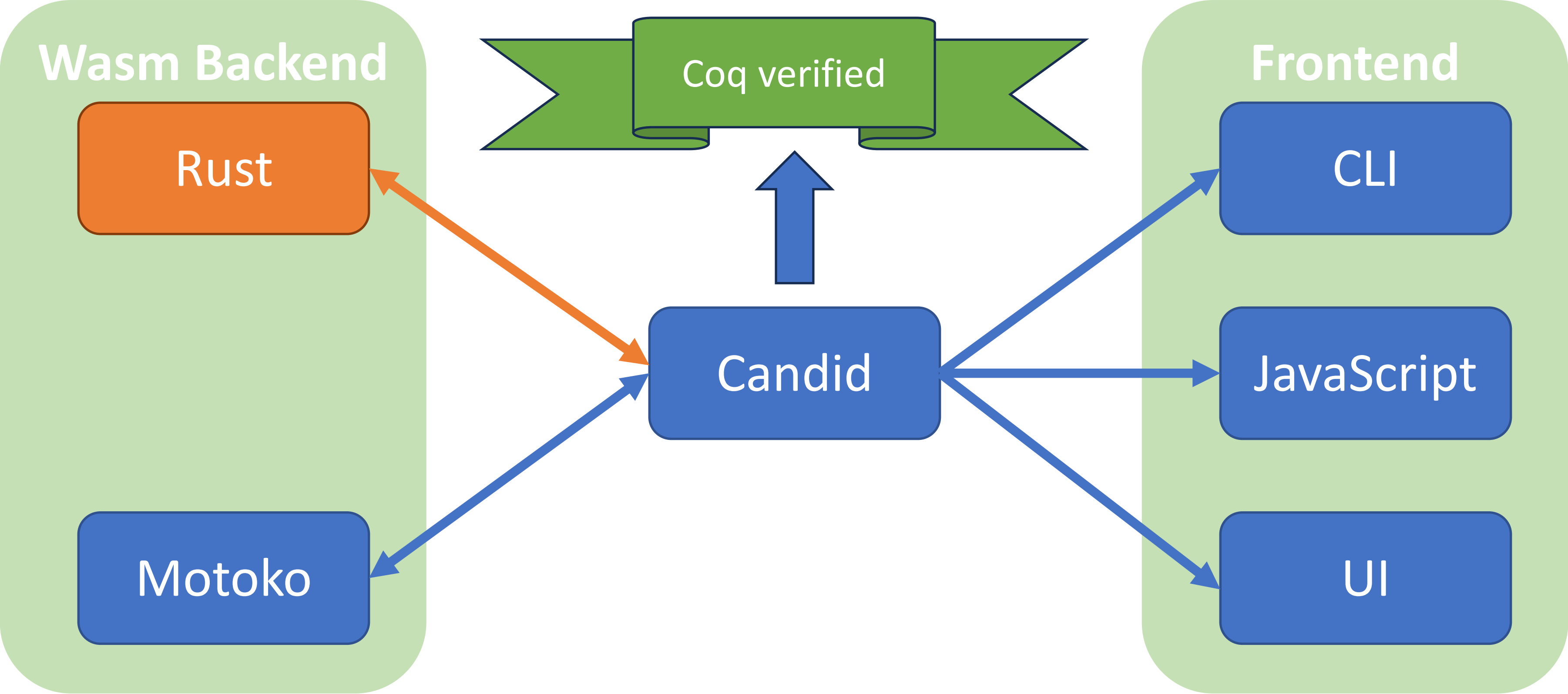
Candid: a strongly typed interface description language

- Primitive types
  - nat, int, nat{8-64}, int{8-64}, float{32,64}, bool, text
- Composite types
  - vec, opt, record, variant
- Reference types
  - func, service
- Supports recursive types
- Structural typing
- Subtyping for upgrade safety

Candid example

```
type tree = variant {
  empty;
  node : record {
    left : tree; val : int; right : tree
  };
};
service : {
  transform :
    (func (int) -> (int), tree) -> (tree);
  getProfile :
    (nat32) -> (record { name : text });
}
```

Candid bindings



Agenda

- Serialize a message
  - How to implement type reflection in Rust
- Export interface description
  - How to simulate monomorphization and share states across procedure macro
- Import interface description
  - How to control generated code

How to serialize a Candid message?

- Candid message contains both **type** and value
- serde can only serialize some of the values

Rust value	serde function	
42 : u8	serialize_u8	✅
Some(42) : Option<i32>	serialize_some	✅
None : Option<i32>	serialize_none	❌
Ok(42) : Result<i8, E>	serialize_struct_variant	❌
vec![42] : Vec<u8>	serialize_seq	✅
vec![] : Vec<u8>	serialize_seq	❌

How to serialize a Candid message?

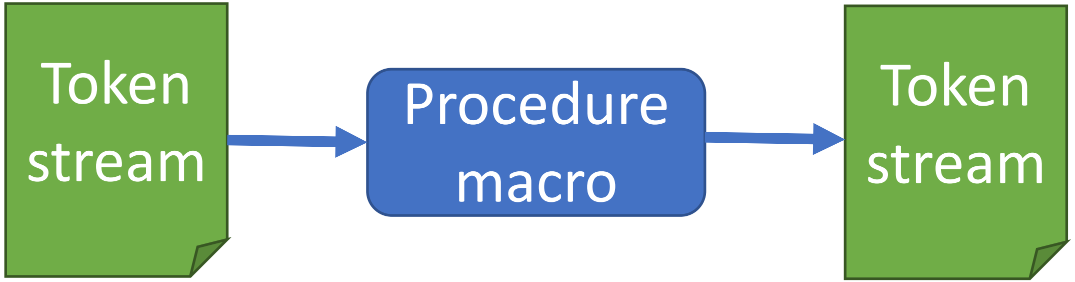
- Candid message contains both **type** and value
- serde can only serialize some of the values
- Similarly, deserialization needs to know the expected Candid type
- Possible solutions
  - Add a new phase in rustc or rust-analyzer
  - Procedure macro

Derive types with procedure macro

- Deriving types from token stream directly

```
#[candid_method]
fn getProfile(id: u32) -> Profile {...}
```

  - No def-use info, cannot handle type aliasing and scoping
  - Types can be generated from other macros
  - Hard to handle recursive types



Token stream

Procedure macro

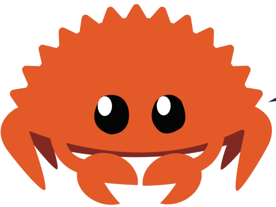
Token stream

Derive types with procedure macro

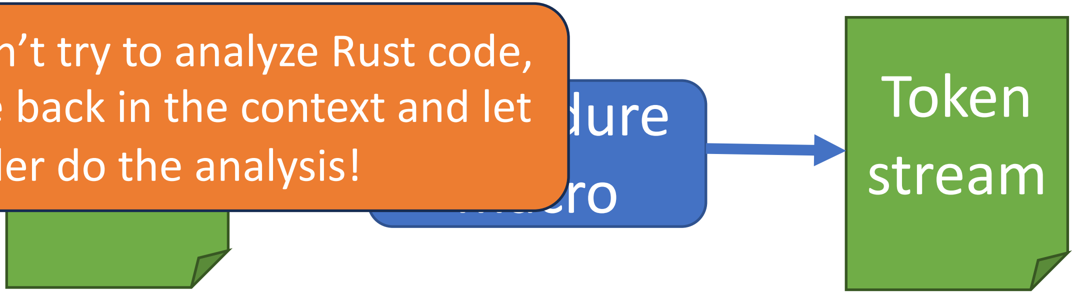
- Deriving types from token stream directly

```
#[candid_method]
fn getProfile(id: u32) -> Profile {...}
```

  - No def-use info, cannot handle type aliasing and scoping
  - Types can be generated from other macros
  - Hard to handle recursive types



Lesson learned: don’t try to analyze Rust code, but to put the code back in the context and let the compiler do the analysis!



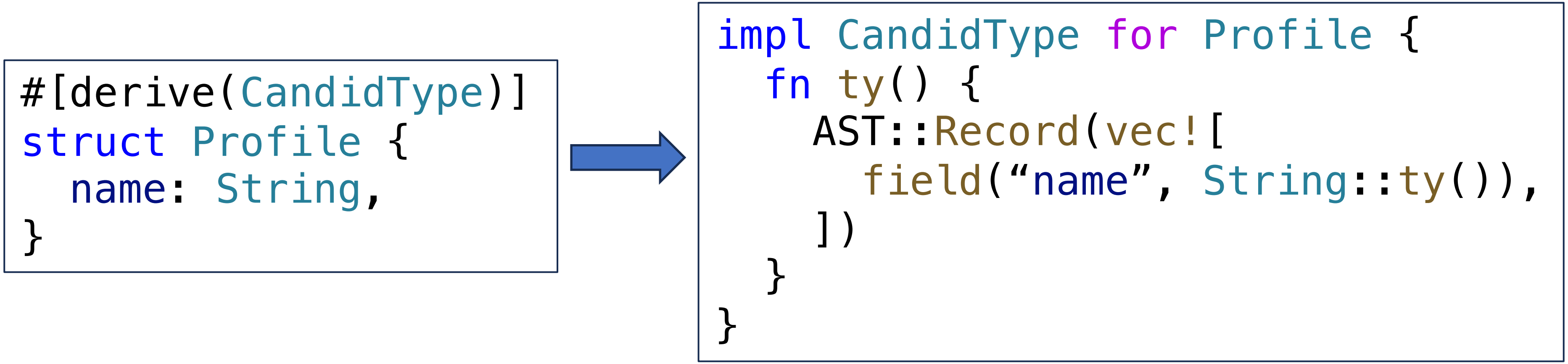
Type reflection in Rust?

- Common pattern: when you don’t know how to implement a feature, define a trait!

```rust
trait CandidType {
  fn ty() -> AST
}
impl CandidType for u8 {
  fn ty() -> AST { AST::Nat8 }
}
impl<T: CandidType> CandidType for Option<T> {
  fn ty() -> AST { AST::Opt(Box::new(T::ty())) }
}
impl<T: CandidType> CandidType for Box<T> {
  fn ty() -> AST { T::ty() }
}
fn typeOf<T: CandidType>(_: &T) -> AST { T::ty() }
```

Derive CandidType for struct/enum

- Use procedure macro to derive CandidType trait



```
#[derive(CandidType)]
struct Profile {
  name: String,
}
```

```
impl CandidType for Profile {
  fn ty() {
    AST::Record(vec![
      field(“name”, String::ty()),
    ])
  }
}
```

What about recursive types?

```
#[derive(CandidType)]
enum Tree {
  Empty,
  Node{left: Box<Tree>, val: Int, right: Box<Tree>},
}
```

```
impl CandidType for Tree {
  fn ty() {
    AST::Variant(vec![
      field(“Empty”, AST::Null),
      field(“Node”, AST::Record(vec![
        field(“left”, Tree::ty()),
        field(“val”, Int::ty()),
        field(“right”, Tree::ty())
      ])),
    ])
  }
}
```

What about recursive types?

- Need a unique identifier for each Rust type
  - `std::any::TypeId::of<T>() -> TypeId`
- Memoize T::ty() with memo table `HashMap<TypeId, AST>`

```
fn ty() -> AST {
  let id = TypeId::of::<Self>();
  if let Some(t) = memo.find(&id) {
    match *t {
      AST::Unknown => AST::Knot(id),
      _ => t.clone(),
    }
  } else {
    memo.insert(id, AST::Unknown);
    let t = Self::_ty();
    memo.insert(id, t.clone());
    t
  }
}
```

```
Tree::ty()
==
AST::Variant(vec![
  field(“Empty”, AST::Null),
  field(“Node”, AST::Record(vec![
    field(“left”, AST::Knot(42)),
    field(“val”, Int::ty()),
    field(“right”, AST::Knot(42)),
  ])),
])
```

What about recursive types?

- Need a unique identifier for each Rust type
  - `std::any::TypeId::of<T>() -> TypeId where T: 'static + ?Sized`
- `'static` because `TypeId::of<'a T>() == TypeId::of::<'b T>()`

```
pub struct TypeId {
  id: usize,
}
impl TypeId {
  pub fn of<T: ?Sized>() -> Self {
    TypeId {
      id: TypeId::of::<T> as usize,
    }
  }
}
```

Recap: Derive types with procedure macro

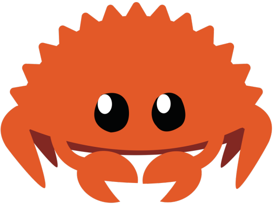
- Deriving types from token stream directly

```
#[candid_method]
fn getProfile(id: u32) -> Profile {...}
```

  - No def-use info, cannot handle type aliasing and scoping
  - Types can be generated from other macros
  - Hard to handle recursive types

Generating code at the same location ensures proper aliasing, scoping, and macro expansion

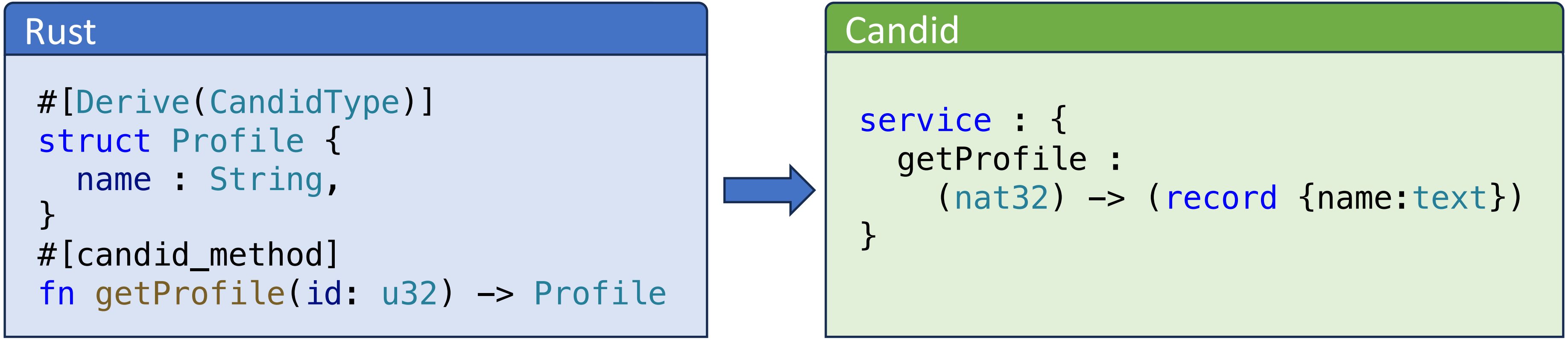
Generated code is often dynamic, e.g., trait, memoization.



Agenda

- Serialize a message
  - How to implement type reflection in Rust
- Export interface description
  - How to simulate monomorphization and share states across procedure macro
- Import interface description
  - How to control generated code

Export service signature to Candid



Rust

```
#[Derive(CandidType)]
struct Profile {
  name : String,
}
#[candid_method]
fn getProfile(id: u32) -> Profile
```

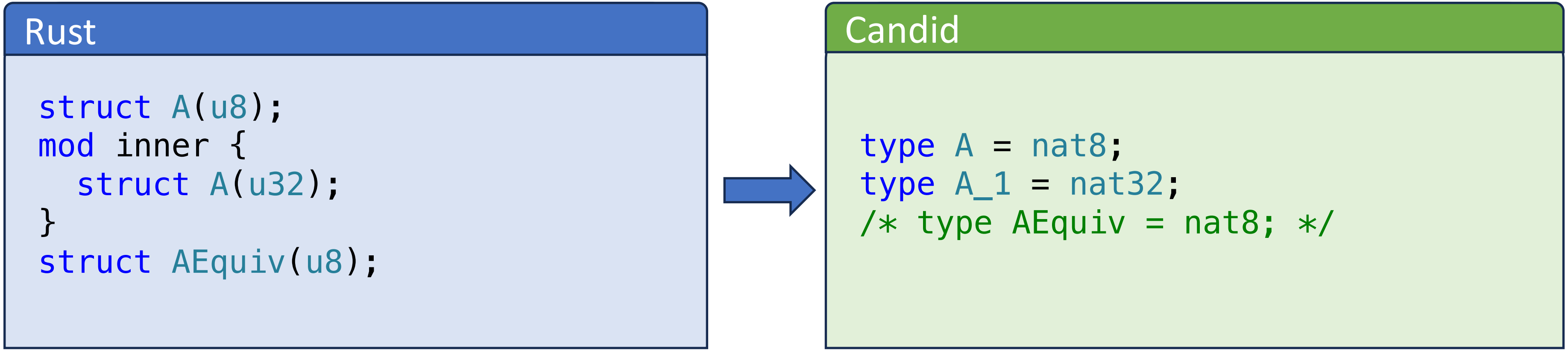
Candid

```
service : {
  getProfile :
    (nat32) -> (record {name:text})
}
```

- Call `T::ty()` for each type
  - All types are inlined, which is not very human readable
  - Doesn’t work for recursive types
- Need a way to export the type bindings

How to export type bindings?

- Old trick: Define an export_type function in CandidType trait to store type bindings needed for each type
  - Type names in different modules/namespaces collapse in Candid



Rust

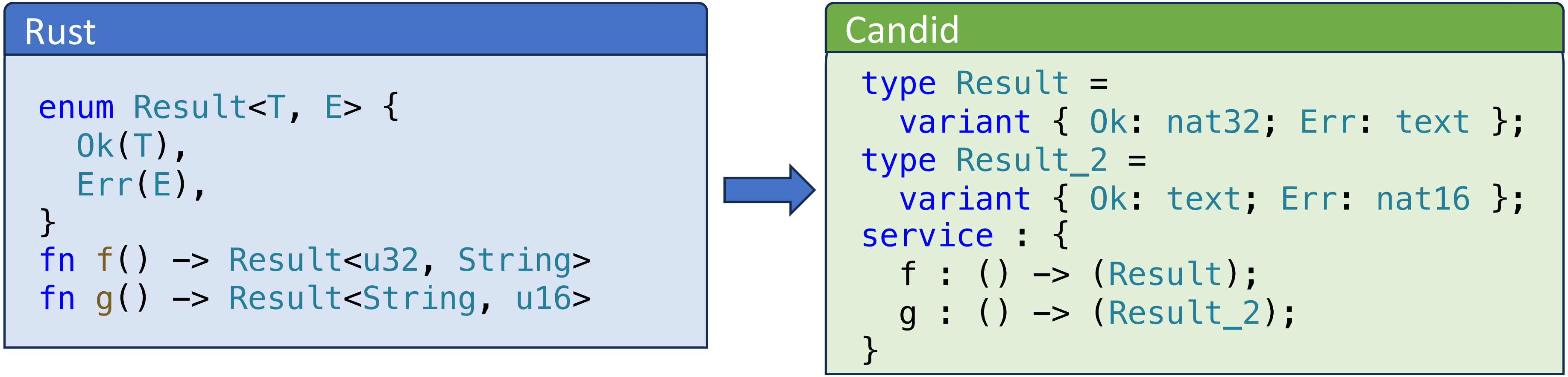
```
struct A(u8);
mod inner {
  struct A(u32);
}
struct AEquiv(u8);
```

Candid

```
type A = nat8;
type A_1 = nat32;
/* type AEquiv = nat8; */
```

How to export type bindings?

- Old trick: Define an export_type function in CandidType trait to store type bindings needed for each type
  - Type names in different modules/namespaces collapse in Candid
  - Polymorphic types only know the instantiated types at the call site



Rust

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
fn f() -> Result<u32, String>
fn g() -> Result<String, u16>
```

Candid

```
type Result =
  variant { Ok: nat32; Err: text };
type Result_2 =
  variant { Ok: text; Err: nat16 };
service : {
  f : () -> (Result);
  g : () -> (Result_2);
}
```

How to export type bindings?

- Old trick: Define an export_type function in CandidType trait to store type bindings needed for each type
  - Type names in different modules/namespaces collapse in Candid
  - Polymorphic types only know the instantiated types at the call site
  - Recursive type AST::Knot(id) needs to convert to a variable name

Candid AST

```
AST::Variant(vec![
  field(“Empty”, AST::Null),
  field(“Node”, AST::Record(vec![
    field(“left”, AST::Knot(42)),
    field(“val”, Int::ty()),
    field(“right”, AST::Knot(42)),
  ])),
])
```

Candid

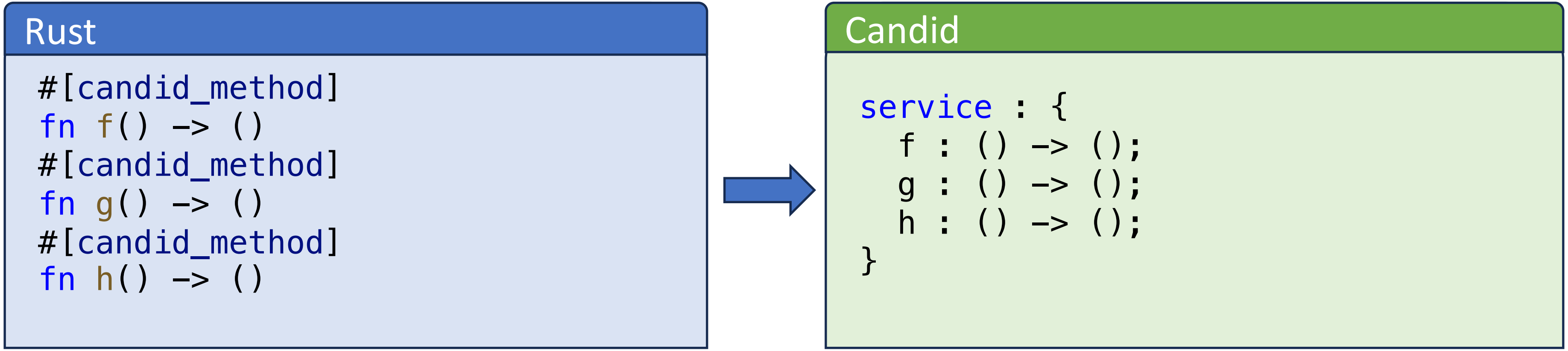
```
type Tree = variant {
  Empty;
  Node : record {
    left: Tree; val: int; right: Tree
  };
};
```

How to export type bindings?

- One Rust type can map to several Candid types (polymorphic types)
  - Inside `T::ty()`, maintain a global `HashMap<TypeId, String>`
    - `fn std::any::type_name<T: ?Sized>() -> &'static str`
    - Append index with duplicate names
- Several Rust types can map to one Candid type (recursive types)
  - Maintain a global `HashMap<AST, TypeId>` to match structurally equivalent types
  - Equivalence checking of recursive types can be done in O(nlogn) time [1]
  - Rewrite `T::ty()` based on the HashMap

[1] N Gauthier, F Pottier. Numbering Matters: First-Order Canonical Forms for Second-Order Recursive Types, ICFP 2004.

Export service signature to Candid



Rust

```rust
#[candid_method]
fn f() -> ()
#[candid_method]
fn g() -> ()
#[candid_method]
fn h() -> ()
```

Candid

```
service : {
  f : () -> ();
  g : () -> ();
  h : () -> ();
}
```

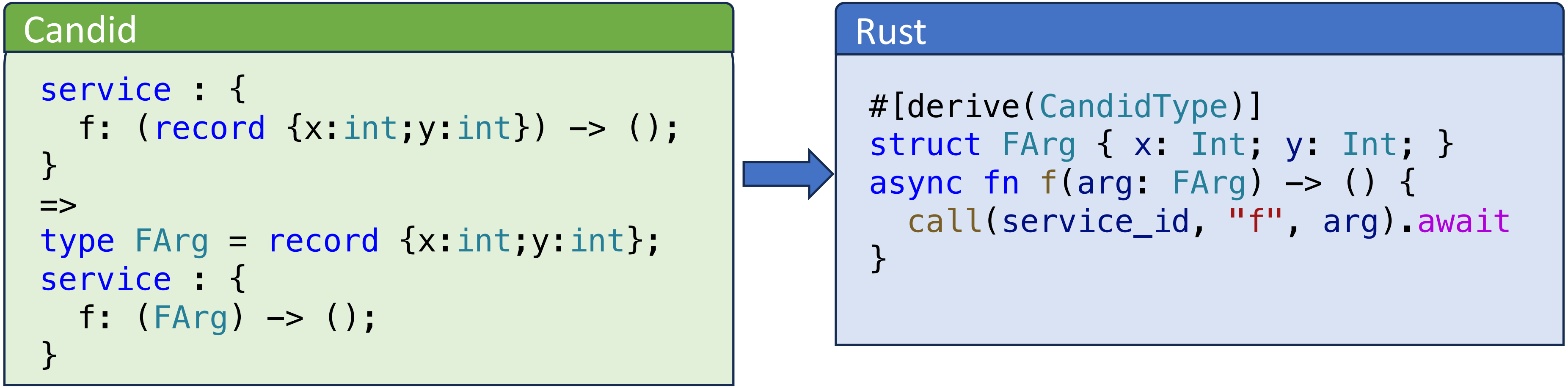
- How to collect method names across attributes?
  - Use `lazy_static!` in the macro (doesn't work for incremental compilation)
  - Custom macro `service! { f : () -> () }` (need to repeat the defs)
  - Attribute macro on impl (recently supported)

Agenda

- Serialize a message
  - How to implement type reflection in Rust
- Export interface description
  - How to simulate monomorphization and share states across procedure macro
- Import interface description
  - How to control generated code

Import external service

- Convert Candid types back to Rust
  - Rewrite AST by converting inlined records/variants into type definitions
  - Identify recursive types and put it into Box



Candid

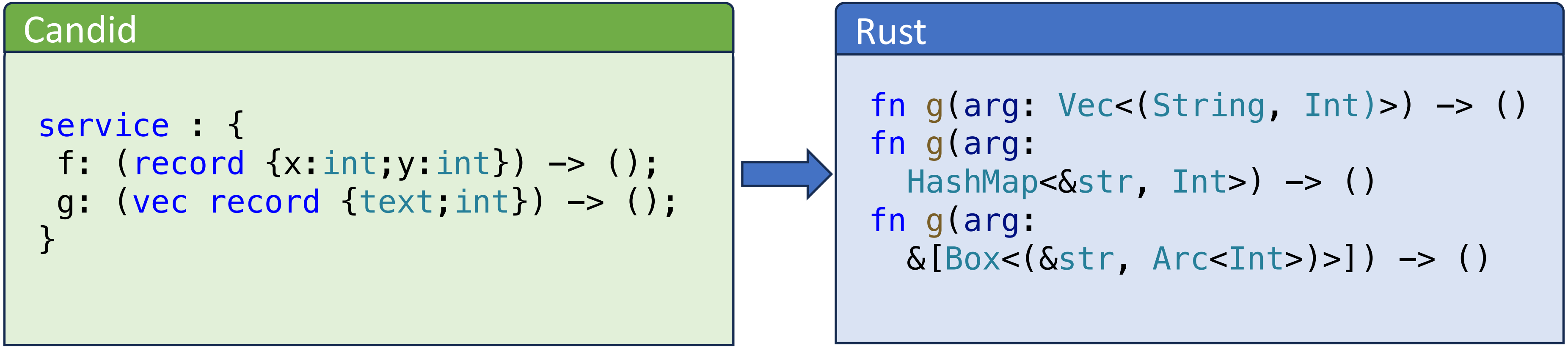
```
service : {
  f: (record {x:int;y:int}) -> ();
}
=>
type FArg = record {x:int;y:int};
service : {
  f: (FArg) -> ();
}
```

Rust

```
#[derive(CandidType)]
struct FArg { x: Int; y: Int; }
async fn f(arg: FArg) -> () {
  call(service_id, "f", arg).await
}
```

Import external service

- Convert Candid types back to Rust
- One-to-many mapping from Candid to Rust types



Candid

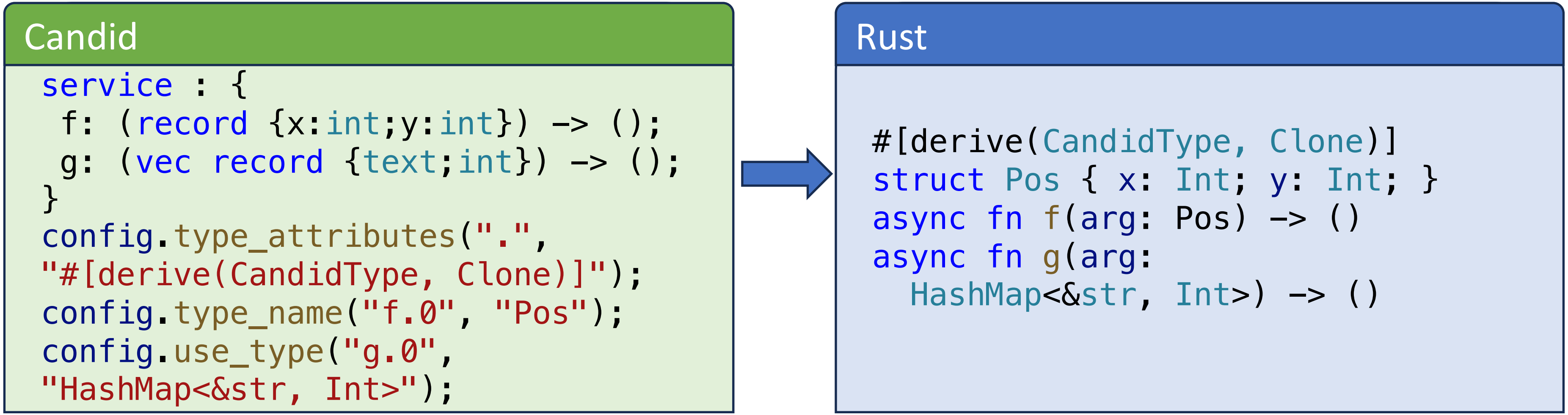
```
service : {
 f: (record {x:int;y:int}) -> ();
 g: (vec record {text;int}) -> ();
}
```

Rust

```
fn g(arg: Vec<(String, Int)>) -> ()
fn g(arg:
  HashMap<&str, Int>) -> ()
fn g(arg:
  &[Box<(&str, Arc<Int>)>]) -> ()
```

Import external service

- Convert Candid types back to Rust
- One-to-many mapping from Candid to Rust types
  - Use a Config struct to control code generation (in progress)



Candid

```
service : {
 f: (record {x:int;y:int}) -> ();
 g: (vec record {text;int}) -> ();
}
config.type_attributes(".",
"#[derive(CandidType, Clone)]");
config.type_name("f.0", "Pos");
config.use_type("g.0",
"HashMap<&str, Int>");
```

Rust

```
#[derive(CandidType, Clone)]
struct Pos { x: Int; y: Int; }
async fn f(arg: Pos) -> ()
async fn g(arg:
  HashMap<&str, Int>) -> ()
```

Summary

- Procedure macro allows us to extend Rust language safely without modifying the compiler
- DFINITY has a Rust SDK to develop smart contracts on the Internet Computer
  - We did all the complicated work, so that developers don’t have to!
- Rust SDK tutorial: https://internetcomputer.org/docs/current/tutorials/
- We have a workshop tonight to demo the Rust SDK!

Thank you！