Session Types in Rust
Reasoning about message passing concurrent programs

Aaron Eline
How Can We Reason about Concurrent/Distributed Code?
Concurrency Primitives

Message Passing *only*

- Extends easily to distributed environment
- Already have “typed channels”
- Doesn’t extend well to more complex types

```go
func main() {
    x := make(chan int)
    x <- "hello"
}
```
doesn’t extend well

- Nothing prevents us from
  - Sending the wrong kind of message
- Forgetting to respond
- Sending the same message twice

```go
type MsgKind int

const (
    helo MsgKind = iota
    ack
    push
    goodbye
)

type msg struct {
    kind MsgKind
    data int
}

x := make(chan msg)
```
Central Questions

• Can we verify that a program implements a specific protocol at compile time?

• Can we do this at low-overhead?

• Can we implement our analysis in a real, industrial-strength language. (In our case: Rust)
The Idea of Session Types

- Can we come up with a type that encapsulates the idea of a protocol?
  - Ie: code that doesn’t implement the protocol right -> won’t type check
- If so, simply create a typed channel with that type
Session Types

• $\epsilon$ - End of protocol
• All other times “sequenced”
• $?$ -> receive
• $!$ -> send
• $\oplus$ -> Pick a branch
• $\&$ -> Receive a branch

$ATM = ?[id]; \oplus\{ok : ATM', err : \epsilon\}$
$ATM' = \&\{ deposit : ?[u64]; ![u64]; \epsilon,$
  
  withdraw : ?[u64]; $\oplus\{ok : \epsilon, err : \epsilon\} \}$
Session Types

• So far: only finite protocols

• Add a recursion operator

\[
\text{ATM'} = \mu t. \& \{ \text{deposit : ？[u64]}; ![u64]; t,} \\
\text{withdraw : ？[u64]; } \oplus \{\text{ok : } t, \text{err : } t\},} \\
\text{quit : } \varepsilon
\]
Dual Types

- Function from type \(\rightarrow\) type computed at compile time
- Computes the opposite protocol
- Ensures client and server match

\[
\begin{align*}
!\tilde{S}; \alpha &= {?\tilde{S}}; \bar{\alpha} \\
?\tilde{S}; \alpha &= {!\tilde{S}}; \bar{\alpha}
\end{align*}
\]

\[
\begin{align*}
\oplus \{l_i: \alpha_i\}_{i \in I} &= \& \{l_i: \bar{\alpha}_i\}_{i \in I} \\
\& \{l_i: \alpha_i\}_{i \in I} &= \oplus \{l_i: \bar{\alpha}_i\}_{i \in I}
\end{align*}
\]

\[
\bar{\epsilon} = \epsilon
\]
Dual Types

\[ \text{ATM} = \?\text{id}; \oplus \{ \text{ok : ATM}', \text{err : } \varepsilon \} \]

\[ \text{ATM}' = \mu t. & \{ \text{deposit : } \?\text{[u64]}; \!\text{[u64]}; t, \]
\[ \text{withdraw : } \?\text{[u64]}; \oplus \{ \text{ok : } t, \text{err : } t \}, \]
\[ \text{quit : } \varepsilon \} \]

\[ \overline{\text{ATM}} = \!\text{id}; \& \{ \text{ok : ATM}', \text{err : } \varepsilon \} \]

\[ \overline{\text{ATM}'} = \mu t. \oplus \{ \text{deposit : } \!\text{[u64]}; \?\text{[u64]}; t, \]
\[ \text{withdraw : } \!\text{[u64]}; \& \{ \text{ok : } t, \text{err : } t \}, \]
\[ \text{quit : } \varepsilon \} \]
Why Rust?

- Rust’s type system is more expressive than those found in most industrial languages
  - Affine Logic
  - Type-Level functions
- Rust is performant and can idiomatically deal with low-level issues
  - Makes it more ideal than some more research-y languages like Haskell or Idris
Affine Types

- A variable who’s type is marked “affine” can only be used once
- Rust uses this rule to track when values can be freed
- We can abuse it to create a protocol type
Affine Types Example

```rust
fn example() -> Foo {
    let f: Foo

    do_something(f)

    return f;
}
```
fn example() -> Foo {
    let f: Foo
    do_something(f)
    return f;
}
Affine Types Example

```rust
def example() -> Foo {
    let f: Foo
    do_something(f)
    return f;
}
```
Affine Types Example

```rust
fn example() -> Foo {
    let f: Foo
    do_something(f)
    return f;
}
```
struct DoOnce;

impl DoOnce {
    fn doThing(self, i: i32) {
    }
}

fn example2() {
    let d = DoOnce {};
    if foo() {
        d.doThing(1);
    } else {
        d.doThing(23);
    }
}

fn do_something(f: Foo) {}
Embedding in Rust

- PhantomData = “Zero Sized Type” ie: completely disappears at compile time, no run time cost
- Thus the “session type” itself completely disappears

```
struct Send<A, P>(PhantomData<(A, P)>)

type Atm = Recv<Id, Choose<Rec<AtmInner>, Eps>>;
type AtmInner = Offer<AtmDeposit, Offer<AtmWithdraw, Quit>>>;

struct Chan<E, P> (Sender<Box<u8>>, Receiver<Box<u8>>, PhantomData<(E, P)>)
```
Case Studies

- ATM implementation (toy)
- Servo implementation
Likes

• Small, simple library
• No external tools
• Type level computation hidden
• Performant implementation in a real language

Dislikes

• Actual primitives are weak
• No large case study