Contextual Metaprogramming for Session Types

Pedro Ângelo, Atsushi Igarashi, **Yuito Murase**, Vasco T. Vasconcelos

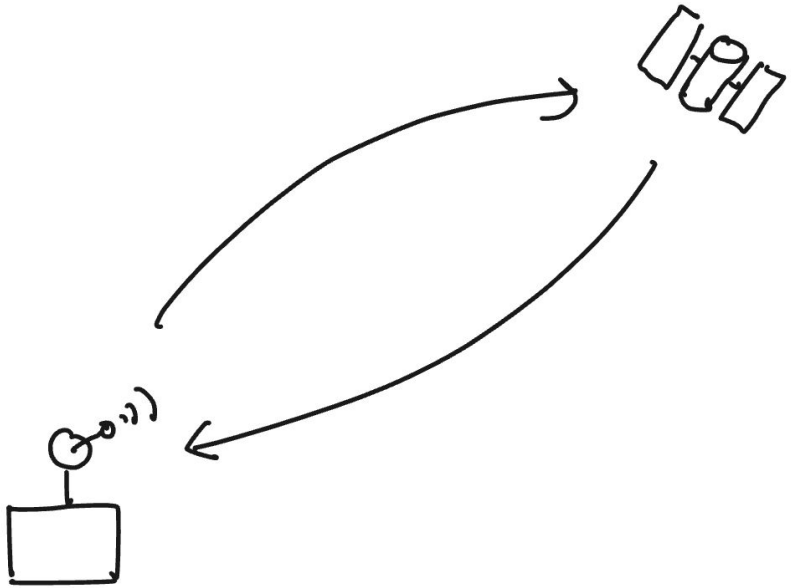
2025-Mar-06 PPL@Gamagori, Aichi

This talk is about...

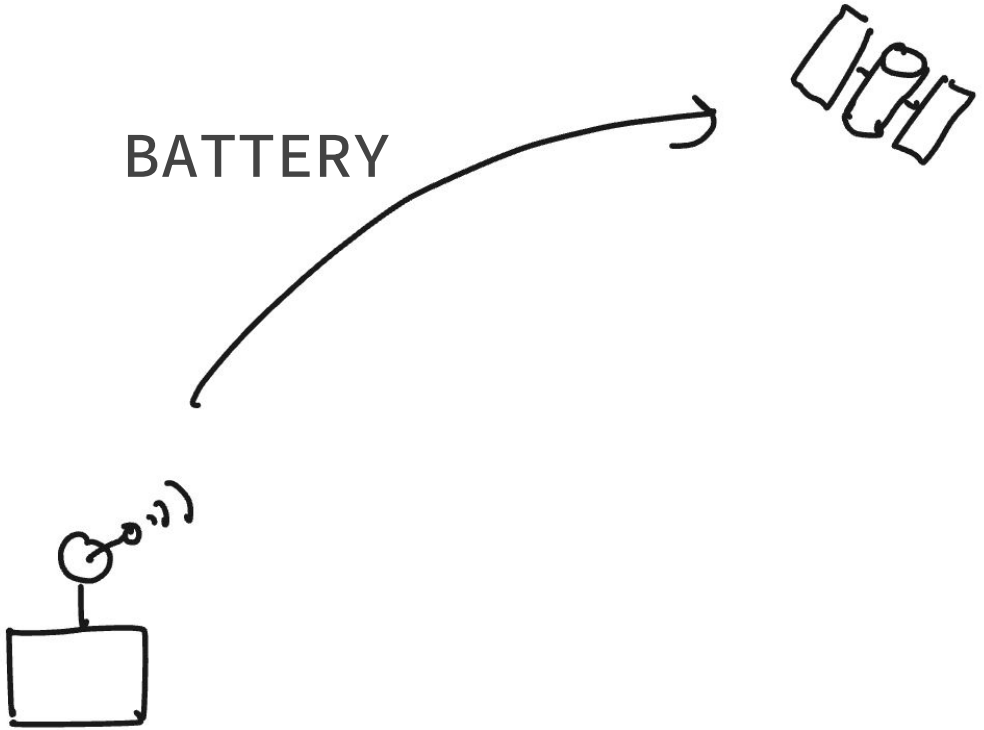
- Combining two paradigms of type systems
  - **Session Types** for distributed programming
  - **Contextual (Modal) Types** for metaprogramming
- Questions would be
  - Why did we choose this combination?
  - What was technical challenges?

Example: Programming Satellites (1)

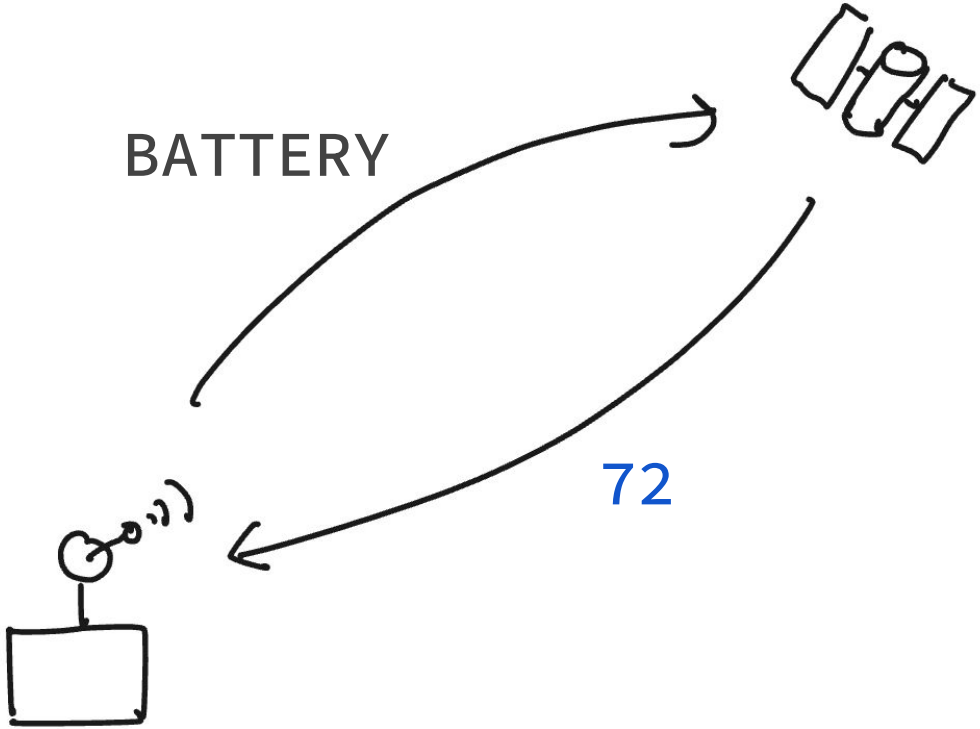
Let us consider programming communication between satellites and control rooms



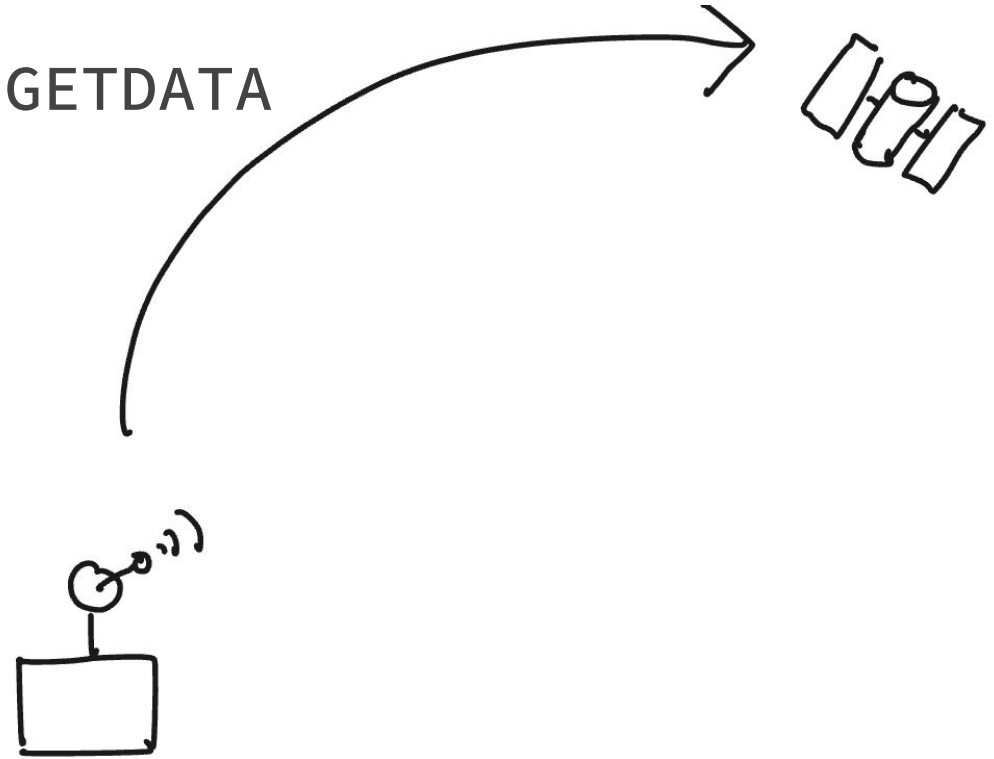
Example: Programming Satellites (2)



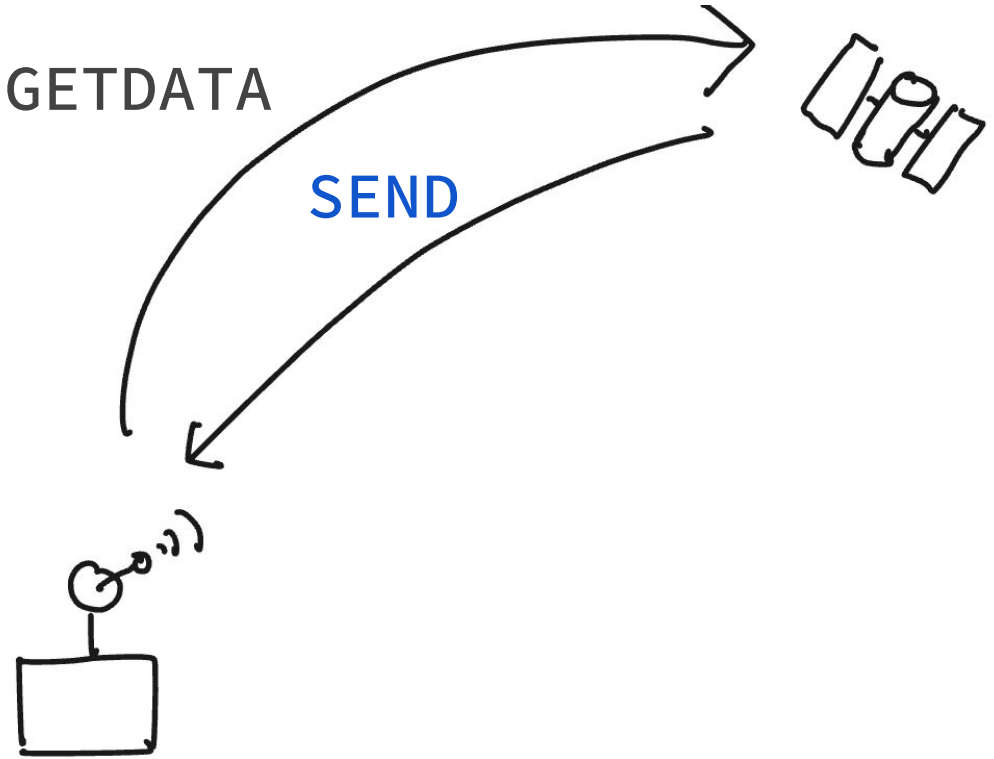
Example: Programming Satellites (2)



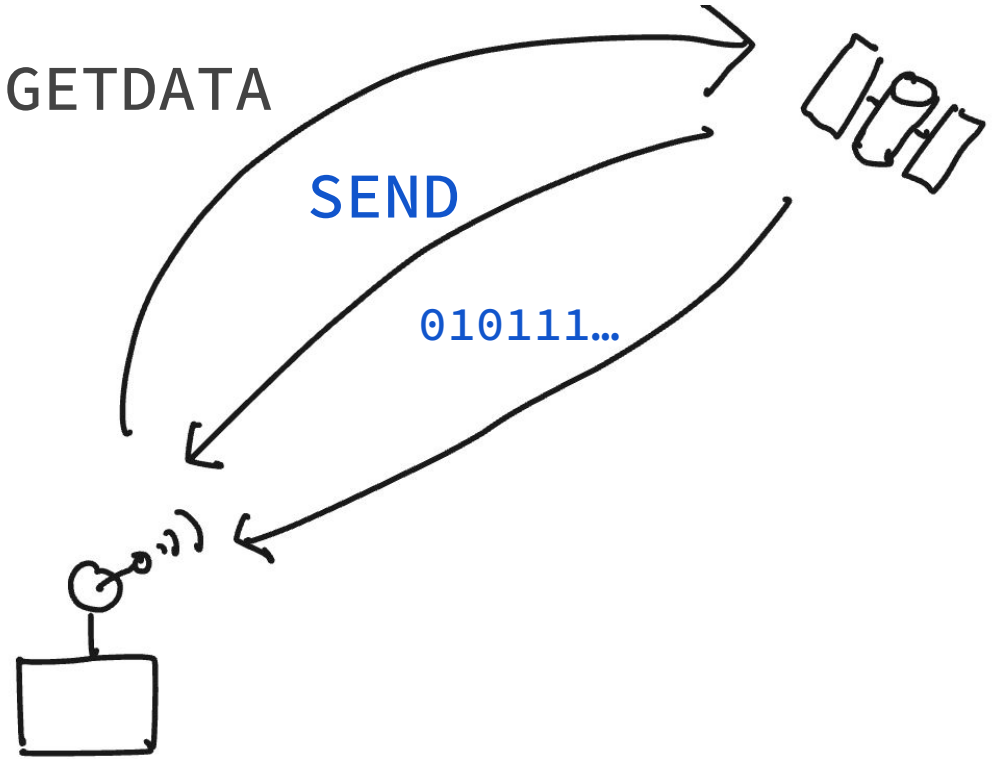
Example: Programming Satellites (3)



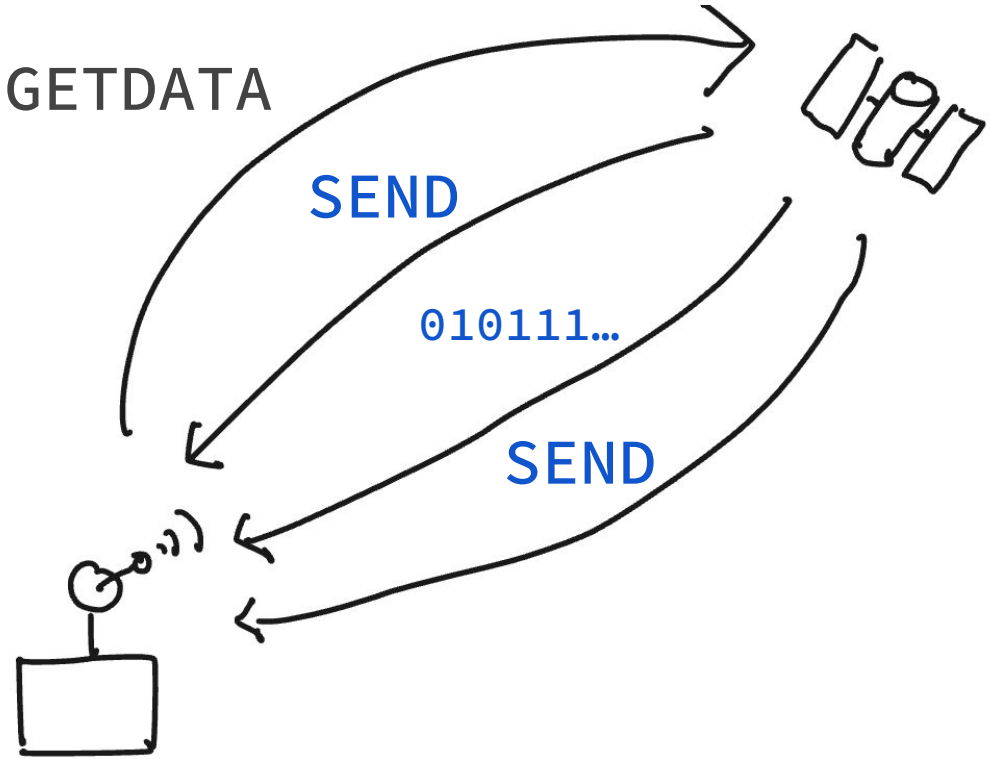
Example: Programming Satellites (3)



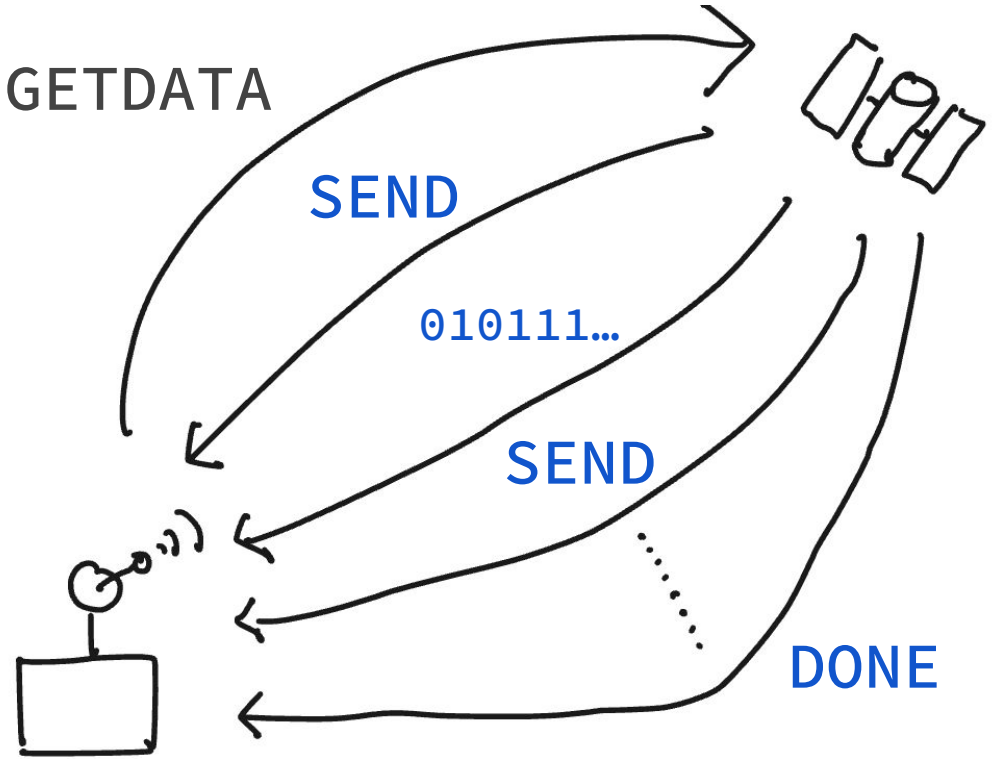
Example: Programming Satellites (3)



Example: Programming Satellites (3)



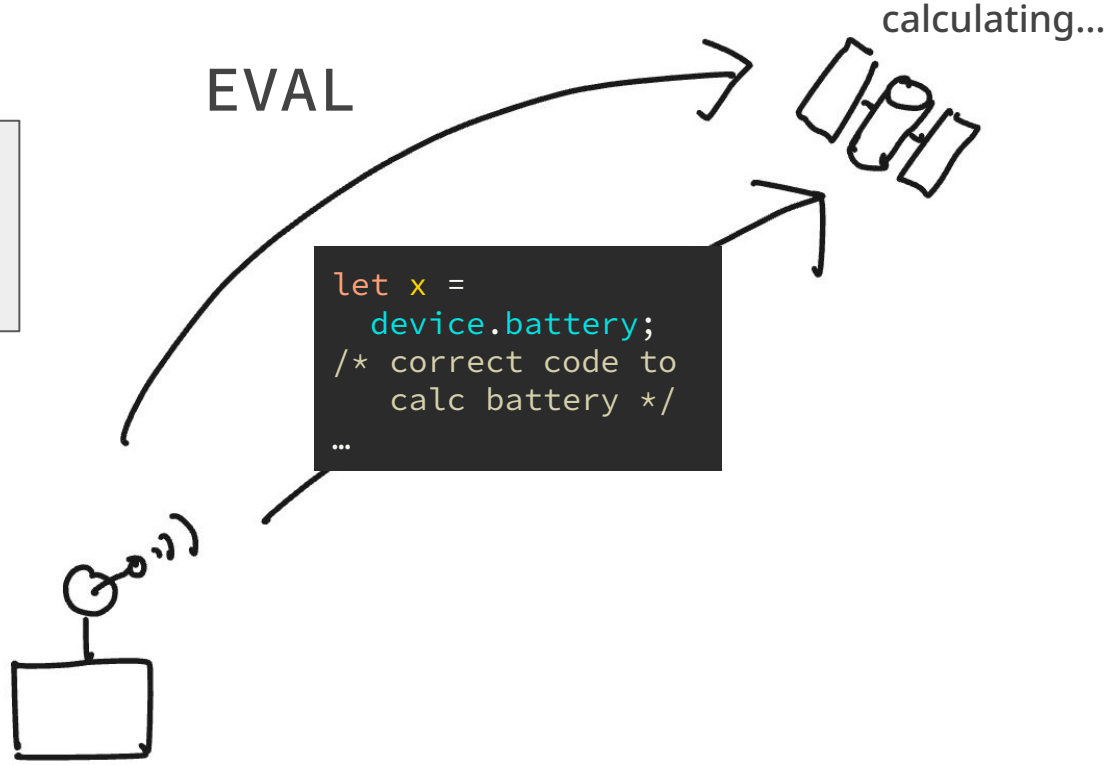
Example: Programming Satellites (3)



Example: Programming Satellites (4)

Found a bug in the satellite-side program!

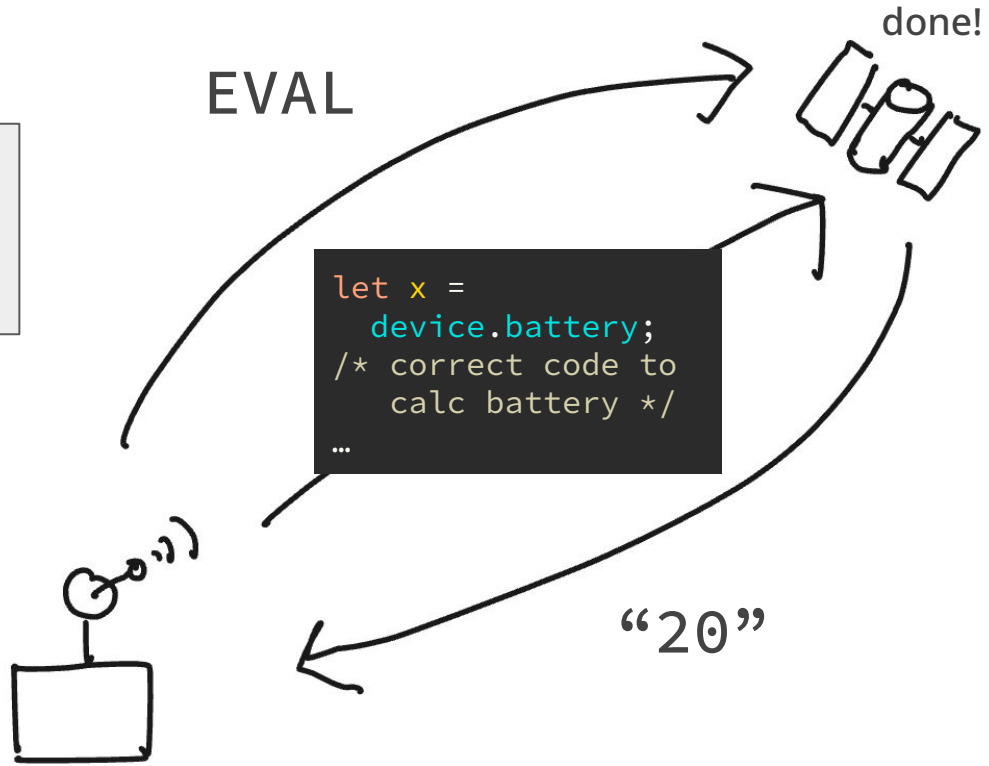
Don’t worry, we can send programs



Example: Programming Satellites (4)

Found a bug in the satellite-side program!

Don’t worry, we can send programs



Communication + Metaprogramming is Common

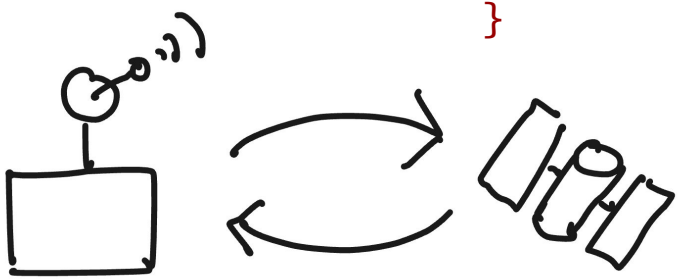
We can observe this pattern in several situations

- Send code to remote devices to run it there
  - e.g. NASA Deep Space 1 **(Search “Lisp in Space”!)**
- Web servers and browsers
  - Server-Side Rendering, Lazy loading, Module Federation, ...
- Volunteer programming (e.g. Folding@Home)
- Hot code reloading

Session Types for Safe Communication

```
type CtrlCh =
rec τ. ⊕{
 BATTERY: ?int.τ,
 GETDATA: rec τ2.
          &{ SEND: ?blob.τ2 ,
             DONE: τ }
 EVAL: !str.?str.τ
}
```

```
type SatelliteCh =
rec τ. &{
 BATTERY: !int.τ,
 GETDATA: rec τ2.
          ⊕{ SEND: !blob.τ2 ,
             DONE: τ }
 EVAL: ?str.!str.τ
}
```

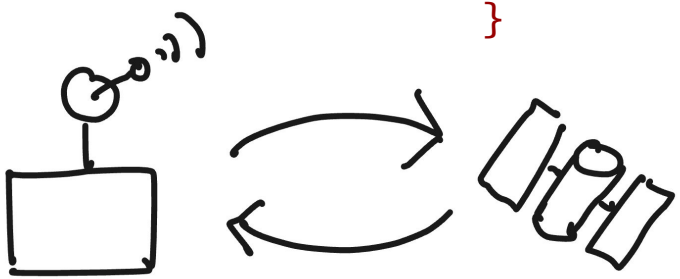


Session Types for Safe Communication

Available commands to send

```
type CtrlCh =
rec τ. ⊕{
 BATTERY: ?int.τ,
 GETDATA: rec τ2.
          &{ SEND: ?blob.τ2 ,
             DONE: τ }
 EVAL: !str.?str.τ
}
```

```
type SatelliteCh =
rec τ. &{
 BATTERY: !int.τ,
 GETDATA: rec τ2.
          ⊕{ SEND: !blob.τ2 ,
             DONE: τ }
 EVAL: ?str.!str.τ
}
```



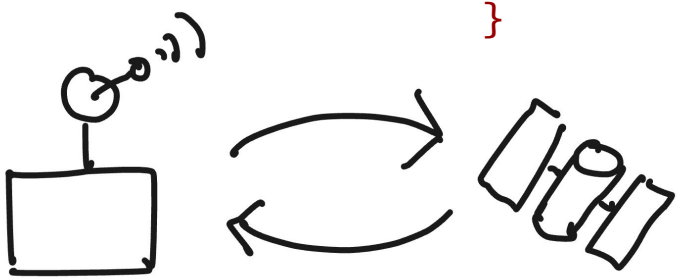
Session Types for Safe Communication

Available commands to send

Receive integer

```
type CtrlCh =
rec τ. ⊕{
 BATTERY: ?int.τ,
 GETDATA: rec τ2.
          &{ SEND: ?blob.τ2 ,
             DONE: τ }
 EVAL: !str.?str.τ
}
```

```
type SatelliteCh =
rec τ. &{
 BATTERY: !int.τ,
 GETDATA: rec τ2.
          ⊕{ SEND: !blob.τ2 ,
             DONE: τ }
 EVAL: ?str.!str.τ
}
```



Session Types for Safe Communication

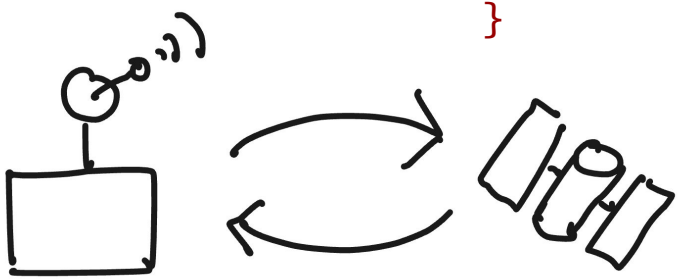
Available commands to send

Receive integer

Recursive type for repetitions

```
type CtrlCh =
rec τ. ⊕{
 BATTERY: ?int.τ,
 GETDATA: rec τ2.
          &{ SEND: ?blob.τ2 ,
             DONE: τ }
 EVAL: !str.?str.τ
}
```

```
type SatelliteCh =
rec τ. &{
 BATTERY: !int.τ,
 GETDATA: rec τ2.
          ⊕{ SEND: !blob.τ2 ,
             DONE: τ }
 EVAL: ?str.!str.τ
}
```



Session Types for Safe Communication

Available commands to send

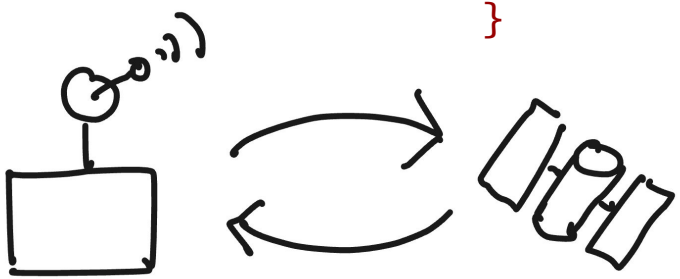
Receive integer

Recursive type for repetitions

```
type CtrlCh =
rec τ. ⊕{
 BATTERY: ?int.τ,
 GETDATA: rec τ2.
          &{ SEND: ?blob.τ2 ,
             DONE: τ }
 EVAL: !str.?str.τ
}
```

Possible commands to receive

```
type SatelliteCh =
rec τ. &{
 BATTERY: !int.τ,
 GETDATA: rec τ2.
          ⊕{ SEND: !blob.τ2 ,
             DONE: τ }
 EVAL: ?str.!str.τ
}
```



Session Types for Safe Communication

Available commands to send

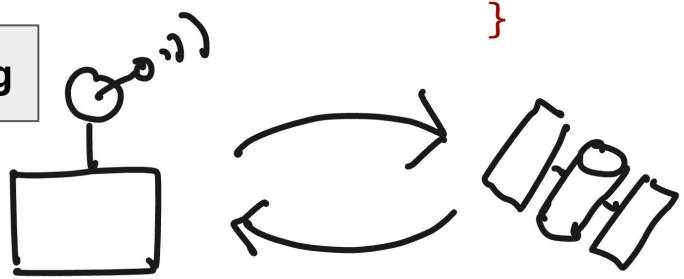
Receive integer

Recursive type for repetitions

```
type CtrlCh =
rec τ. ⊕{
 BATTERY: ?int.τ,
 GETDATA: rec τ2.
          &{ SEND: ?blob.τ2 ,
             DONE: τ }
 EVAL: !str.?str.τ
}
```

Send string

Possible commands to receive



```
type SatelliteCh =
rec τ. &{
 BATTERY: !int.τ,
 GETDATA: rec τ2.
          ⊕{ SEND: !blob.τ2 ,
             DONE: τ }
 EVAL: ?str.!str.τ
}
```

Session Types for Safe Communication

Available commands to send

Receive integer

Recursive type for repetitions

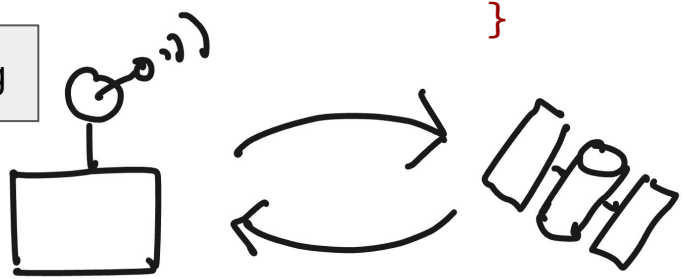
```
type CtrlCh =
rec τ. ⊕{
 BATTERY: ?int.τ,
 GETDATA: rec τ2.
          &{ SEND: ?blob.τ2 ,
             DONE: τ }
 EVAL: !str.?str.τ
}
```

Send string

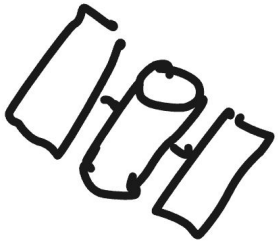
Possible commands to receive

Dual

```
type SatelliteCh =
rec τ. &{
 BATTERY: !int.τ,
 GETDATA: rec τ2.
          ⊕{ SEND: !blob.τ2 ,
             DONE: τ }
 EVAL: ?str.!str.τ
}
```



Channels are Linear Resources



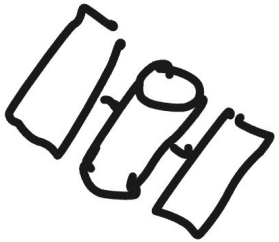
```
type SatelliteCh =
rec τ. &{
 BATTERY: …
 GETDATA: …
 EVAL: ?str.!str.τ
}
```

```
let rec serve(ch: SatelliteCh): unit =
  match ch with
    case BATTERY => …
    case GETDATA => …
    case EVAL => \ch -> {
      let (code, ch) = receive ch in
      let ch = send ch (exec code) in
      serve ch
    }
```

Channels must be used exactly once

⇒ **Linear Types**

Channels are Linear Resources



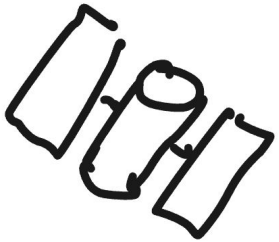
```
type SatelliteCh =
rec τ. &{
 BATTERY: …
 GETDATA: …
 EVAL: ?str.!str.τ
}
```

```
let rec serve(ch: SatelliteCh): unit =
  match ch with
    case BATTERY => …
    case GETDATA => …
    case EVAL => \ch -> {
      let (code, ch) = receive ch in
      let ch = send ch (exec code) in
      serve ch
    }
```

Channels must be used exactly once

⇒ **Linear Types**

Channels are Linear Resources



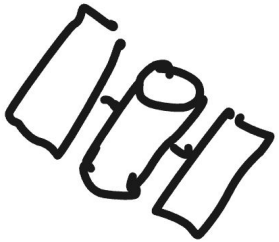
```
type SatelliteCh =
rec τ. &{
 BATTERY: …
 GETDATA: …
 EVAL: ?str.!str.τ
}
```

```
let rec serve(ch: SatelliteCh): unit =
  match ch with
    case BATTERY => …
    case GETDATA => …
    case EVAL => \ch -> {
      let (code, ch) = receive ch in
      let ch = send ch (exec code) in
      serve ch
    }
```

Channels must be used exactly once

⇒ **Linear Types**

Channels are Linear Resources



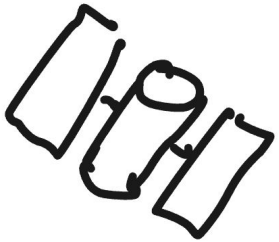
```
type SatelliteCh =
rec τ. &{
 BATTERY: …
 GETDATA: …
 EVAL: ?str.!str.τ
}
```

```
let rec serve(ch: SatelliteCh): unit =
  match ch with
    case BATTERY => …
    case GETDATA => …
    case EVAL => \ch -> {
      let (code, ch) = receive ch in
      let ch = send ch (exec code) in
      serve ch
    }
```

Channels must be used exactly once

⇒ **Linear Types**

Channels are Linear Resources



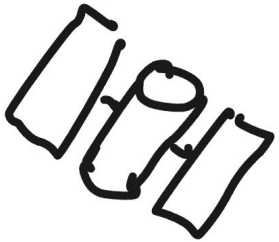
```
type SatelliteCh =
rec τ. &{
 BATTERY: …
 GETDATA: …
 EVAL: ?str.!str.τ
}
```

```
let rec serve(ch: SatelliteCh): unit =
  match ch with
    case BATTERY => …
    case GETDATA => …
    case EVAL => \ch -> {
      let (code, ch) = receive ch in
      let ch = send ch (exec code) in
      serve ch
    }
```

Channels must be used exactly once

⇒ **Linear Types**

Contextual Types for Metaprogramming



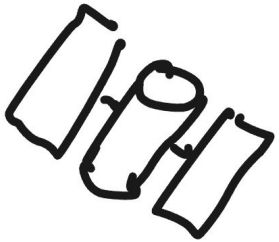
```
type SatelliteCh =
rec τ. &{
 …
 EVAL: ?str.!str.τ
}
```

Programs as string

```
…
case EVAL => \ch -> {
  let (code, ch) = receive ch in
  let ch = send ch (exec code) in
  serve ch
}
```

It is dangerous to evaluate code fragments

Contextual Types for Metaprogramming



```
type SatelliteCh =
rec τ. &{
 …
 EVAL: ?[Γ⊢str].!str.τ
}
```

Contextual Type
[Nanevski+ 08][Jang+ 22]

```
…
case EVAL => \ch -> {
  let (code, ch) = receive ch in
  let box U = code in
  let ch = send ch U[device,...] in
  serve ch
}
```

Execute code with supplying resources

Resources that a program requires (i.e. *context*)

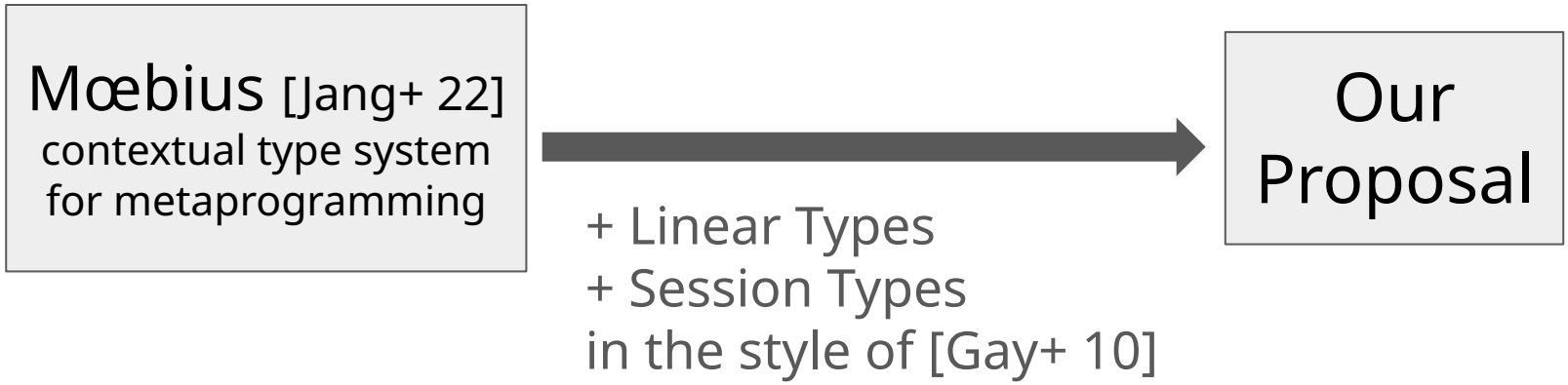
$[\Gamma \vdash str]$

Result of the computation

Example

```
`{x,y. x+y}
:[int,int⊢int]
```

What We Did



We formalized a typed programming language
that provide both session types and contextual types

- Type System
- Operational Semantics

Code is ALWAYS unrestricted resource

We determine linearity of values a la Walker [2005]

```
1000
“some string”
```

Can be copied
⇒ **unrestricted**

```
(ch: CtrlCh)
```

Channels are
linear

Code is ALWAYS unrestricted resource

We determine linearity of values à la Walker [2005]

```
1000
“some string”
```

Can be copied ⇒ **unrestricted**

```
(ch: CtrlCh)
```

Channels are **linear**

```
λx.send ch x
```

Depends on a linear resource ⇒ **linear**

```
λch.send ch 1
```

Independent from linear rsc. ⇒ **unrestricted**

Code is ALWAYS unrestricted resource

We determine linearity of values a la Walker [2005]

```
1000
“some string”
```

Can be copied
⇒ **unrestricted**

```
(ch: CtrlCh)
```

Channels are
linear

```
λx.send ch x
```

Depends on a linear resource ⇒ **linear**

```
λch.send ch 1
```

Independent from linear rsc. ⇒ **unrestricted**

```
`{x,ch.send ch x}
```

Our type system enforce that code is closed (= independent from any resources)
⇒ unrestricted

Safety Properties

We confirmed that our proposal satisfies expected safety properties

Theorem 3 (Progress for evaluation)**.** *Let* $\Gamma^S \vdash M : T$*. Then,*

1. M *is a value, or*
2. $M \to N$*, for some* N*, or*
3. M *is of the form* $E[N]$*, for some* E*, with* N *of one of the following forms:* close x, wait x, send $v\ x$, receive x, select $l\ x$, match x with $\{l \to N_l\}^{l \in L}$, new v *or* fork v.

Theorem 4 (Absence of immediate errors)**.** *If* $\vdash P$ *then* P *is not a runtime error.*

Related Work

- Modal types for distributed programs[Jia+04][Murphy+04]
  - Assume simpler communication with multiple processes
- Multi-tier programming
  - e.g., Hop.js[Serrano+16], Links[Cooper+07]
  - Experimental Programming Language Implementations for web applications
- Multi-stage computation
  - We want to evaluate code at runtime
    ⇒ type systems like λ○[Davies17] are not adequate

We are Seeking for Future Directions

Currently, novelty in our proposal seems a bit weak, while we believe that Communication+Metaprogramming have better chemistry...

- Different-Style communications from modal calculi
- Extension towards multi-tier programming
- Staged Semantics in distributed programming

Let us know if you have good idea!

References (1/2)

- Nanevski, A., Pfenning, F., & Pientka, B. (2008). Contextual modal type theory. ACM Transactions on Computational Logic, 9(3), 23:1-23:49. https://doi.org/10.1145/1352582.1352591
- Jang, J., Gélineau, S., Monnier, S., & Pientka, B. (2022). Mœbius: Metaprogramming using contextual types: the stage where system f can pattern match on itself. Proceedings of the ACM on Programming Languages, 6(POPL), 1–27. https://doi.org/10.1145/3498700
- Gay, S. J., & Vasconcelos, V. T. (2010). Linear type theory for asynchronous session types. Journal of Functional Programming, 20(1), 19–50. https://doi.org/10.1017/S0956796809990268
- Walker, D. (2005) Substructural type systems. In Advanced Topics in Types and Programming Languages, Pierce, B. C. (ed.). MIT Press, Cambridge, Massachusetts, Chapter 1, pp. 3–43.

References (2/2)

- Jia, L., & Walker, D. (2004). Modal Proofs as Distributed Programs. In D. Schmidt (Ed.), Programming Languages and Systems (pp. 219–233). Springer. https://doi.org/10.1007/978-3-540-24725-8_16
- Murphy, T., Crary, K., Harper, R., & Pfenning, F. (2004). A symmetric modal lambda calculus for distributed computing. Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004., 286–295. https://doi.org/10.1109/LICS.2004.1319623
- Serrano, M., & Prunet, V. (2016). A glimpse of Hopjs. SIGPLAN Not., 51(9), 180–192. https://doi.org/10.1145/3022670.2951916
- Cooper, E., Lindley, S., Wadler, P., & Yallop, J. (2007). Links: Web Programming Without Tiers. Formal Methods for Components and Objects, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12
- Davies, R. (2017). A Temporal Logic Approach to Binding-Time Analysis. Journal of the ACM, 64(1), 1:1-1:45. https://doi.org/10.1145/3011069