Types for Protocols

Peter Thiemann

University of Freiburg

Summer BOB, August 2019

Table of Contents

Outline

Types

- A success story since [Church 1940]
- Most frequently used formal method
- Invented to
  - describe successful computations
  - prevent run-time errors

Errors Prevented by Traditional Types

Avoid data being used differently than intended

- A bit pattern intended as a floating point number should not be used as an integer

$\Rightarrow$ Hence, Float and Int should be distinct types!

- A bit pattern intended as an integer should not be used as an address (of a string)

$\Rightarrow$ Hence, String and Int should be distinct types!

Traditional Type Systems

- This kind of type system is extremely well researched
- Put into practice in many statically typed programming languages
- **Eliminate a whole class of errors**

A Typical Type Language

$$
\begin{aligned}
T, U ::= {} & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Float} \\
& \mid (T, U) \mid T + U \mid [T] \\
& \mid \{\ell_i : T_i\} \mid [\ell_i : T_i] \mid T \to U
\end{aligned}
$$

For example

- 42 : Int
- True : Bool
- 6.022E23 : Float
- (True, 1) : (Bool, Int)

But we can find more errors than that!

- Many of them are still in the scope of a type system

Track additional properties of values

- refined types (e.g., subsets of numbers or strings)
- data integrity and confidentiality → security type systems
- units of measure
- etc

But we can find entirely different errors, too!

Track behaviors — behavioral types

- Values / objects have a **state**
- Changes over time in response to external stimuli

The good old file example

```
module File : sig
  type t

  val fopen : path → t
  val write : t → string → unit
  val close : t → unit
end
```

- f = fopen "foo" creates a new file named `foo` for writing
- The file handle f has an *abstract type* File.t
- We can write f "..." arbitrary many times and then **close** f
- We still have a hold on f, but writing again yields an error!

```
let f = fopen "foo" in
let _ = write f "stuff" in
let _ = close f in
let _ = write f "more" in (* run-time error *)
```

A simplistic solution

```
module File : sig
  type t : lin

  val fopen : path → t
  val write : t → string → t
  val close : t → unit
end
```

- We only change the interface to file handles
- The type `File.t` of file handles is now *linear*

⇒ cannot be deleted or duplicated

- `write` returns a fresh file handle to the updated file
- **close** consumes the file handle
- Writing after close is a type error:

```
let f1 = fopen "foo" in
let f2 = write f1 "stuff" in
let _ = close f2 in
let _ = write f2 "more" in (* type error *)
```

On linear typing

- every variable (of linear type) must be used exactly once
- rooted in linear logic [Girard 1987]
- has found uses in memory management and more generally in resource management

Outline

Types for protocols — session types

- Types for structured bidirectional communication
- Session types prescribe
  1. the values transmitted
     classical type safety
  2. the direction and sequencing of transmissions
     session fidelity
- Session types codify the structure of communication and make it available to reasoning and programming tools

A little history

- Session types were born more than 25 years ago
- Originally stated for the π-calculus, a calculus for communication
- Seminal papers
  - Kohei Honda, “Types for Dyadic Interaction”, CONCUR 1993.
  - Takeuchi, Honda & Kubo, “An Interaction-Based Language and its Typing System”, PARLE 1994.
  - Honda, Vasconcelos & Kubo, “Language Primitives and Type Discipline for Structured Communication-Based Programming”, ESOP 1998.
- Presentation influenced by
  Simon Gay, Vasco Vasconcelos, ”Linear Type Theory for Asynchronous Session Types”, Journal of Functional Programming 20(1):19-50 (2010).

The good old math server

Server type

```
type Server = &{
  Neg: ?Int. !Int. Server,
  Add: ?Int. ?Int. !Int. Server,
  Quit: end}
```

The good old math server

Server type

```
type Server = &{
  Neg: ?Int. !Int. Server,
  Add: ?Int. ?Int. !Int. Server,
  Quit: end}
```

Client type

```
type Client = ⊕{
  Neg: !Int. ?Int. Client,
  Add: !Int. !Int. ?Int. Client,
  Quit: end}
```

The good old math server

Server type

```
type Server = &{
  Neg: ?Int. !Int. Server,
  Add: ?Int. ?Int. !Int. Server,
  Quit: end}
```

Client type

```
type Client = ⊕{
  Neg: !Int. ?Int. Client,
  Add: !Int. !Int. ?Int. Client,
  Quit: end}
```

Duality

```
Client = dualof Server
```

Session types

$S ::=$	
$\&\{\ell_i : S_i\}$	branch / offer / external choice
$\oplus\{\ell_i : S_i\}$	select / internal choice
$?T.S$	input T continue as S
$!T.S$	output T continue as S
end	marks the end of the protocol
$T ::= S \mid Int \mid * \mid T \otimes T \mid T \to T \mid \ldots$	functional fragment

- the "." indicates sequencing
- `Neg`, `Add`, `Quit` are *choice labels*, which are all different

Math server implementation

Server type

```
type Server = &{
  Neg: ?Int. !Int. Server,
  Add: ?Int. ?Int. !Int. Server,
  Quit: end}
```

Math server implementation

Server type

```
type Server = &{
  Neg: ?Int. !Int. Server,
  Add: ?Int. ?Int. !Int. Server,
  Quit: end}
```

Implementation

```
server : Server → Unit
server c =
  rcase c of
    Neg → c. let x, c = recv c
                 c = send c (−x) in
             server c
    Add → c. let x, c = recv c
                 y, c = recv c
                 c = send c (x + y) in
             server c
```

Zooming in on changing types

```
server : Server → Unit
server c =
  rcase c of
    Neg → c. // c : ?Int. !Int. Server
             let x, c = recv c
             // c : !Int.Server
                    c = send c (−x) in
             // c : Server
             server c
    Add → c. // c : ?Int. ?Int. !Int. Server
             let x, c = recv c
             // c : ?Int. !Int. Server
                 y, c = recv c
             // c : !Int. Server
                    c = send c (x + y) in
             // c : Server
             server c
    Quit → c. close c
```

. . . and a client

```
negClient : dualof Server → Int
negClient d x =
  let     d = select Neg d
          d = send d x
       r, d = recv d
          d = select Quit d in
  r
```

Making a connection

```
ports
  p : #Server

let s = accept p in
    server s
||
let c = request p in
    negclient c 42
```

- #Server is the type of a port that can spawn off new sessions with endpoints of type Server and **dualof** Server
- accept obtains the session of type Server
- request obtain the session of the dual type Client
- accept and request synchronize on the port

Key points

- Session endpoints are *linear*: each endpoint occurs exactly once in a system
- Session types *change* with each communication
- Structure of the code matches structure of the session type
- Sessions are *higher-order*,
  i.e., session endpoints may be transmitted

Outline

Outline

Deadlocks

```
ports
  p1 : #(!Int.end)
  p2 : #(!Int.end)

let s1 = accept p1
    s2 = accept p2
    s1 = send s1 41   -- stuck
    s2 = send s2 42
||
let c1 = request p1
    c2 = request p2
    v2, c2 = receive c2   -- stuck
    v1, c1 = receive c1
```

- first-order sessions (only base types transmitted)
- deadlock because synchronous send operation blocks

Deadlocks

```
type S = !Int. end
ports
  p1 : #(?S. end)
  p2 : #S

let s1 = accept p1
    s2 = accept p2
    c2, s1 = receive s1
    close s1
    v2, c2 = receive c2  -- stuck
    s2 = send s2 42
||
let c1 = request p1
    c2 = request p2
    c1 = send c1 c2
in close c1
```

- higher-order sessions (c2 is sent over c1)
- first process is stuck even if sending is asynchronous

Deadlocks

- Session types (in general) do *not* rule out deadlocks
- But there are versions that do
  - Based on cycle detection [Kobayashi] [Padovani]
  - Based on topological constraints [Caires, Pfenning] [Wadler]
- Topological constraints are enforced by linking process creation with session creation

Outline

Flexibility — Subtyping

- Following Liskov's substitution principle [Liskov, Wing 1994]: "if $S <: T$, then it is safe to use a value of type S where a value of type T is expected"
- The implementation does not have to match the type of the port exactly
- it can implement a *supertype*, that is, the port's type is more restricted
- There are two sources of subsumption
  - external choice: a session of type $\&\{\ell_1 : S_1, \dots, \ell_n : S_n\}$ can be used even when *more* choices are expected
  - internal choice: a session of type $\oplus\{\ell_1 : S_1, \dots, \ell_n : S_n\}$ can be used with any subset of the given choices

Flexibility — Subtyping

Example: the client

```
type Client = ⊕{
  Neg: !Int. ?Int. Client,
  Add: !Int. !Int. ?Int. Client,
  Quit: end}
```

but the actual code does not use the Add choice:

```
type Client1 = ⊕{
  Neg: !Int. ?Int. Client1,
  Quit: end}
```

or completely aligned with the code

```
type Client2 = ⊕{
  Neg: !Int. ?Int. ⊕{
  Quit: end}}
```

The types are related by subtyping: Client <: Client1 <: Client2

More sources of subtyping

- If a session $!T.S$ is ready to send a value of type T, we can also send a value of a *subtype* $T' <: T$.
- If a session $?T.S$ is ready to receive a value of type T, we can also expect a value of a *supertype* $T' :> T$.
- Analogous to subtyping for functions.
- First study:
  Simon J. Gay, Malcolm J. Hole, "Types and Subtypes for Client-Server Interactions", ESOP1999, 74-90

More sources of subtyping

- Implicit assumption so far: synchronous communication
- But session types are also sound for asychronous communication!
- Asynchrony gives further scope for subtyping because the sender can keep sending even when the receiver is not catching up immediately

Example

Synchronous version

```
negClient :
  dualof Server → Int
negClient d x =
  let    d = select Neg d
         d = send d x
      r, d = recv d
         d = select Quit d
  in r
```

Asynchronous version

```
asyncNegClient :
  ??? → Int
asyncNegClient d x =
  let    d = select Neg d
         d = send d x
         d = select Quit d
      r, d = recv d
  in r
```

- In the asyncNegClient we have

  ```
  d : ⊕{ Neg: !Int. ⊕{ Quit: ?Int. end }}
  ```

  which is *not* a supertype of **dualof** Server
- but it would be an *asynchronous supertype*

The state of subtyping

- Synchronous subtyping is decidable
- (Unrestricted) asychronous subtyping is undecidable
- State of the art:
  - Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, Gianluigi Zavattaro: A Sound Algorithm for Asynchronous Session Subtyping (extended version). CoRR abs/1907.00421 (2019)
  - Julien Lange, Nobuko Yoshida: On the Undecidability of Asynchronous Session Subtyping. FoSSaCS 2017: 441-457
  - Mario Bravetti, Marco Carbone, Gianluigi Zavattaro: On the boundary between decidability and undecidability of asynchronous session subtyping. Theor. Comput. Sci. 722: 19-51 (2018)
  - Mario Bravetti, Marco Carbone, Gianluigi Zavattaro: Undecidability of asynchronous session subtyping. Inf. Comput. 256: 300-320 (2017)

Outline

Extension: Exceptions and timeouts

Exceptions

- In a realistic setting, network connections do not work flawlessly
- Session types can be extended to deal with such disruptions in an orderly way
- Simon Fowler, Sam Lindley, J. Garrett Morris, Sára Decova: Exceptional asynchronous session types: session types without tiers. PACMPL 3(POPL): 28:1-28:29 (2019)

Timeouts

- Session types can deal with timeouts by adding extra timed choices to external choices
- Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, Nobuko Yoshida: Asynchronous Timed Session Types - From Duality to Time-Sensitive Processes. ESOP 2019: 583-610

Extension: Gradual typing

- Gradual typing allows programmers to leave parts of types unspecified, but to retain type safety by inserting suitable run-time checks
- For session types, graduality requires checking adherence to linear use of sessions as well as session fidelity dynamically at run time.
- Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, Philip Wadler: Gradual session types. PACMPL 1(ICFP): 38:1-38:28 (2017)

Other approaches to run-time monitoring

- Compilation to timed automata
  Rumyana Neykova, Laura Bocchi, Nobuko Yoshida: Timed runtime monitoring for multiparty conversations. Formal Asp. Comput. 29(5): 877-910 (2017)
- Session type contracts
  Hernán C. Melgratti, Luca Padovani: Chaperone contracts for higher-order sessions. PACMPL 1(ICFP): 35:1-35:29 (2017)

Outline

Extension: Dependent types

- Many practical protocol have variable-length fields
- The naive encoding in a session type relies on a list-like protocol structure:

```
type Bytes = &{
  More: ?Byte. Bytes,
  Done: end }
```

- This type enables sending an arbitrary number of Bytes, but it is inefficient due to the intervening “flow control” messages More and Done.

Extension: Dependent types (2)

- It would be more efficient to be able to send the number n of bytes first, followed by exactly n bytes without any administrative messages.
- A typical scenario for dependent types
- To this end, we need to
  - write a (type-level) function from numbers to session types
  - write a dependently typed function that actually receives the byte stream
- To simplify matters, we return a list of Bytes, but we could also return a suitably sized vector.

Extension: Dependent types (3)

```
type B n = if n == 0
  then end
  else ?Byte. B(n-1)

type NBytes = ?(n:Nat). B n

readBytes' : (n: Nat) → B n → list Byte
readBytes' n c = if n == 0
  then []
  else let v, c = receive c
           vs = readBytes' (n-1) c
       in  v :: vs

readBytes : NBytes → list Byte
readBytes c =
  let n, c = receive c in
  readBytes' n c
```

Extension: Dependent types (4)

Challenges

- Types for sending and receiving must admit dependency
- Implies the need for Π and Σ types
  (dependent products and sums)
- Type checking and subtyping need to be decidable
- Type-level functions (like B) need to be terminating

Outline

Extension: Multiparty session types

Binary session types

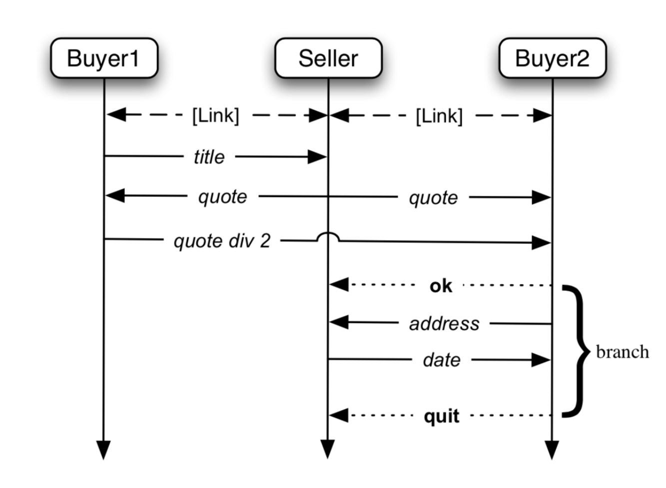
- Binary session types describe communication between two partners
- A single process may have several sessions, but communication on them is not coordinated and can lead to deadlock.

Multiparty session types [Honda, Yoshida, Carbone: POPL 2008]

- Communication between several processes is governed by a single *global type*
- Global type can be analyzed to guarantee deadlock freedom
- Each process communicates according to its *local type* which is projected from the global type
- Local type checking sufficient to guarantee communication safety

Multiparty session types (2)

Buyer-seller example from [Honda et al 2008]



Multiparty session types (3)

Global type for buyer-seller

1. $B1 \to S$: title.
2. $S \to B1$: quote.
3. $S \to B2$: quote.
4. $B1 \to B2$: quote.
5. $B2 \to S$: $\begin{cases} ok: & B2 \to S & : address. \\ & S \to B2 & : date.end \\ quit: & end \end{cases}$

Multiparty session types (4)

Local type for B1

$$S!\mathit{title}.S?\mathit{quote}.B2!\mathit{quote}$$

Local type for B2

$$S?\mathit{quote}.B1?\mathit{quote}.S \oplus \{\mathit{ok} : S!\mathit{address}.S?\mathit{date}.\mathit{end}, \mathit{quit} : \mathit{end}\}$$

Multiparty session types (4)

Local type for B1

$$S!title.S?quote.B2!quote$$

Local type for B2

$$S?quote.B1?quote.S \oplus \{ok : S!address.S?date.end, quit : end\}$$

- Local type checking as for binary session types

Multiparty session types (4)

Local type for B1

$$S!title.S?quote.B2!quote$$

Local type for B2

$$S?quote.B1?quote.S \oplus \{ok : S!address.S?date.end, quit : end\}$$

- Local type checking as for binary session types
- Conditions on global type guarantee independance of participating processes

Outline

Conclusion

- powerful formalism to model protocols
- context π-calculus and concurrent λ-calculus
- reasonable implementations in several languages (Java, Scala, OCaml, Haskell, etc), but none has all guarantees
- related to contracts, type state, etc
- many extensions

Further reading

S. J. Gay and A. Ravara (editors). Behavioural Types: from Theory to Tools. River Publishers, 2017.

Thank you!