# context-free session type inference

Luca Padovani

ESOP 2017

# outline

# one-slide introduction to session types

## Definition (session type)

- type-level specification of a communication protocol
- how a session endpoint is meant to be used

$$T = {?}\mathbf{int}\,;{?}\mathbf{int}\,;{!}\mathbf{bool}\,;\mathbf{end}$$

$$\overline{T} = {!}\mathbf{int}\,;{!}\mathbf{int}\,;{?}\mathbf{bool}\,;\mathbf{end}$$

$$S = \&[\mathtt{Eq} : T, \mathtt{Neg} : {?}\mathbf{int}\,;{!}\mathbf{int}\,;\mathbf{end}]$$

## Property (safety & fidelity)

*Well-typed programs communicate safely and respect protocols*

# a non-uniform object

with FuSe [Padovani ’17]

```
let stack c =
  let rec none u = (* empty stack *)
    match branch u with
    | Push u → let x, u = receive u in
               none (some x u)
    | Stop u → u
  and some y u =   (* stack with y on top *)
    match branch u with
    | Push u → let x, u = receive u in
               some y (some x u)
    | Pop u → send y u
  in none c
```

# a non-uniform object

with FuSe [Padovani '17]

```
let stack c =
  let rec none u = (* empty stack *)
    match branch u with
    | Push u → let x, u = receive u in
               none (some x u)
    | Stop u → u                        ☠ dead code
  and some y u =   (* stack with y on top *)
    match branch u with
    | Push u → let x, u = receive u in
               some y (some x u)
    | Pop u → send y u                  ☠ dead code
  in none c
```

$$\mathsf{c} : \mu X.\&[\mathsf{Push} : ?\alpha ; X]$$

tv.pdf

# from ordinary to **context-free** session types

## Ordinary session types

- sequential composition limited to prefixes $?\alpha; S$
- language of (finite) traces is regular

## Context-free session types [Thiemann & Vasconcelos '16]

- general form of sequential composition $T; S$
- language of (finite) traces is context-free
- typability++, precision++

$$
\begin{aligned}
X &= \&[\texttt{Push} : ?\alpha; Y; X,\ \texttt{Stop} : \textbf{end}] \\
Y &= \&[\texttt{Push} : ?\alpha; Y; Y,\ \texttt{Pop} : !\alpha]
\end{aligned}
$$

# a context-free session type system

## Key ingredients

[Thiemann & Vasconcelos '16]

- monoidal laws for sequential composition

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash e : S} \; T \sim S$$

- polymorphic recursion

## Conclusion

- type checking is substantially **more difficult** (open problem)
- library implementation is challenging if at all possible

## Compromise

- give up some flexibility (**ask the programmer** for help!)
- **enable** context-free session type **inference**

# outline

# **resuming** finished sub-protocols

1. Distinguish finished protocols
   - **end**: finished protocol (no more actions afterwards)
   - **done**: finished sub-protocol (must be **resumed** later)

2. Use sequential composition for
   - ordering actions **in types**
   - structuring **code**

$$f \qquad u$$

# **resuming** finished sub-protocols

1. Distinguish finished protocols
   - **end**: finished protocol (no more actions afterwards)
   - **done**: finished sub-protocol (must be **resumed** later)

2. Use sequential composition for
   - ordering actions **in types**
   - structuring **code**

$T \rightarrow \textbf{done}$ ▷ $f$ $u$ ◁ $T\,;S$

# **resuming** finished sub-protocols

1. Distinguish finished protocols
   - **end**: finished protocol (no more actions afterwards)
   - **done**: finished sub-protocol (must be **resumed** later)

2. Use sequential composition for
   - ordering actions **in types**
   - structuring **code**

$T \to \textbf{done}$ $f$ @> $u$ $T\,;S$

$(T \to \textbf{done}) \to T\,;S \to S$

# the idea is **flawed**

$$f : T \rightarrow \textbf{done}$$

- $f$ takes an endpoint $u$ of type $T$
- $f$ returns an endpoint $v$ of type **done**
- $v$ is **not necessarily** the same as $u$
- `@>` could be used for casting $v$ to an arbitrary $S$

# session types with endpoint **identities**

$$u \;:\; [T]_{\varrho}$$

# session types with endpoint **identities**

$$u \ : \ [T]_\varrho$$

$$f \ : \ [T]_\varrho \to [\textbf{done}]_\varrho$$

# session types with endpoint **identities**

$$u \;:\; [T]_\varrho$$

$$f \;:\; [T]_\varrho \to [\textbf{done}]_\varrho$$

$$\mathsf{send} \;:\; t \to [\,!\,t\,;T]_\varrho \to [T]_\varrho$$

# session types with endpoint **identities**

$$u \;:\; [T]_\varrho$$

$$f \;:\; [T]_\varrho \to [\textbf{done}]_\varrho$$

$$\mathsf{send} \;:\; t \to [\,!\,t\,;T]_\varrho \to [T]_\varrho$$

$$\mathtt{@>} \;:\; ([T]_\varrho \to [\textbf{done}]_\varrho) \to [T\,;S]_\varrho \to [S]_\varrho$$

# session types with endpoint **identities**

$$u \;:\; [T]_\varrho$$

$$f \;:\; [T]_\varrho \to [\textbf{done}]_\varrho$$

$$\texttt{send} \;:\; t \to [\,!\,t\,;T]_\varrho \to [T]_\varrho$$

$$\texttt{@>} \;:\; ([T]_\varrho \to [\textbf{done}]_\varrho) \to [T\,;S]_\varrho \to [S]_\varrho$$

$$\texttt{create} \;:\; \textbf{unit} \to \exists\varrho.([T]_\varrho \times [\overline{T}]_{\overline{\varrho}})$$

# outline

# GV with resumable sessions

GV [Gay & Vasconcelos '10]

- CBV $\lambda$-calculus
- threads
- session primitives `create`, `send`, `receive`, ...

Existentials

- pack, unpack

A difficulty with subject reduction

- @> is operationally irrelevant
- @> affects the type of $u$ for an unknown number of reductions

$$f \mathbin{@\!>} u \quad \rightarrow \quad ?$$

# resumptions as a **bracketing** construct

$$\{ \, f \; u \, \}_u$$

# resumptions as a **bracketing** construct

$$\{\, f\ u \,\}_u$$

$$\frac{}{u : [T\,; S]_\iota \vdash \{e\}_u :}$$

# resumptions as a **bracketing** construct

$$\{\, f\ u\, \}_u$$

$$\frac{u : [T]_\iota \vdash e :}{u : [T\,;S]_\iota \vdash \{e\}_u :}$$

# resumptions as a **bracketing** construct

$$\{\, f\ u \,\}_u$$

$$\frac{u : [T]_\iota \vdash e : [\mathbf{done}]_\iota}{u : [T\,;S]_\iota \vdash \{e\}_u :}$$

# resumptions as a **bracketing** construct

$$\{\, f\ u \,\}_u$$

$$\frac{u : [T]_\iota \vdash e : [\textbf{done}]_\iota}{u : [T\,; S]_\iota \vdash \{e\}_u : [S]_\iota}$$

# subject reduction with resumptions

$$
\begin{array}{lll}
u : & [T_1 ; S]_\iota & \{e_1\}_u \\
 & & \downarrow_* \\
u : & [T_2 ; S]_\iota & \{e_2\}_u \\
 & & \downarrow_* \\
u : & [T_3 ; S]_\iota & \{e_3\}_u \\
 & & \downarrow_* \\
 & & \vdots \\
 & & \downarrow_* \\
u : & [\textbf{done} ; S]_\iota & \{u\}_u \\
 & & \downarrow \\
u : & [S]_\iota & u
\end{array}
$$

# properties of the type system

Well-typed programs are well behaved

- communication safety
- protocol fidelity (`@>` guarantees sequentiality)

Identity **uniqueness** is key requirement

$$\begin{array}{rcl} \text{ouch}\ (x, y) & \to^* & (x, y) \\ \text{ouch} & : & [\textbf{done}\,; T]_\iota \times [\textbf{done}\,; S]_\iota \to [S]_\iota \times [T]_\iota \end{array}$$

- if two endpoints have the same identity safety is compromised
- if two **peers** have the same identity **fidelity** is compromised

# outline

# endpoint type encoding

$$[T]_\iota \rightsquigarrow (t_1, t_2, \iota, \bar{\iota})\ \mathtt{t}$$

## Property (duality as equality)

- *If* $[T]_\iota \rightsquigarrow (t, s, \iota, \bar{\iota})\ \mathtt{t}$ *then* $[\overline{T}]_{\bar{\iota}} \rightsquigarrow (s, t, \bar{\iota}, \iota)\ \mathtt{t}$

# session creation with **first-class modules**

$\texttt{val create : unit} \to \exists\varrho.([T]_\varrho \times [\overline{T}]_{\overline{\varrho}})$

```
val create : unit → (module Package)
```

```
module type Package = sig
  type i and j (* abstract identities *)
  val unpack : unit → (α,β,i,j) t × (β,α,j,i) t
end
```

```
let module S = (val create ()) in
let u, v = S.unpack () in (* only once! *)
fork server u; client v
```

# endpoint resumption

```
val (@>) : ([α]ϱ → [done]ϱ) → [α;β]ϱ → [β]ϱ

let (@>) = Obj.magic
```

# the stack, resumed

```
let stack c =
  let rec none u =
    match branch u with
    | Push u → let x, u = receive u in
               none (some x    u)
    | Stop u → u
  and some y u =
    match branch u with
    | Push u → let x, u = receive u in
               some y (some x    u)
    | Pop u → send y u
  in none c
```

# the stack, resumed

```
let stack c =
  let rec none u =
    match branch u with
    | Push u → let x, u = receive u in
               none (some x @> u)    (* resume *)
    | Stop u → u
  and some y u =
    match branch u with
    | Push u → let x, u = receive u in
               some y (some x @> u) (* resume *)
    | Pop u → send y u
  in none c
```

# the stack protocol **inferred**

```
val stack : ([< 'Stop of β | 'Push of (((γ msg, 0, δ, ε) t,
(((([< 'Pop of (((0, γ msg, δ, ε) t, (1, 1, δ, ε) t) seq,
((γ msg, 0, ε, δ) t, (1, 1, ε, δ) t) seq, δ, ε) t |
'Push of (((γ msg, 0, δ, ε) t, (((φ, 0, δ, ε) t, (φ, 0,
δ, ε) t) seq, ((0, φ, ε, δ) t, (0, φ, ε, δ) t) seq, δ,
ε) t) seq, ((0, γ msg, ε, δ) t, (((0, φ, ε, δ) t, (0,
φ, ε, δ) t) seq, ((φ, 0, δ, ε) t, (φ, 0, δ, ε) t) seq,
ε, δ) t) seq, δ, ε) t ] as ψ) tag as φ, 0, δ, ε) t, (α,
0, δ, ε) t) seq, ((0, φ, ε, δ) t, (0, α, ε, δ) t) seq,
δ, ε) t) seq, ((0, γ msg, ε, δ) t, (((0, φ, ε, δ) t,
(0, α, ε, δ) t) seq, ((φ, 0, δ, ε) t, (α, 0, δ, ε) t)
seq, ε, δ) t) seq, δ, ε) t ] tag as α, 0, δ, ε) t → β
```

$$
\begin{aligned}
\texttt{stack} &: [X]_\varrho \to [\beta]_\varrho \\
X &= \&[\texttt{Push} : ?\gamma ; Y ; X,\ \texttt{Stop} : \beta] \\
Y &= \&[\texttt{Push} : ?\gamma ; Y ; Y,\ \texttt{Pop} : !\gamma]
\end{aligned}
$$

# tree serialization



```
type α tree = Leaf | Node of α × α tree × α tree

let rec send_tree t u =
  match t with
  | Leaf → select 'Leaf u
  | Node (x, l, r) → let u = select 'Node u in
                     let u = send x u in
                     let u = send_tree l u in
                     send_tree r u
```

$$\begin{aligned} \texttt{send\_tree} &: \alpha\,\texttt{tree} \to [X]_\varrho \to [X]_\varrho \\ X &= \oplus[\texttt{Leaf} : X, \texttt{Node} : !\alpha ; X] \end{aligned}$$

# tree serialization with resumption

```
type α tree = Leaf | Node of α × α tree × α tree

let rec send_tree t u =
  match t with
   | Leaf → select 'Leaf u
   | Node (x, l, r) → let u = select 'Node u in
                       let u = send x u in
                       let u = send_tree l @> u in
                       send_tree r u
```

$$\texttt{send\_tree} \;:\; \alpha\ \texttt{tree} \to [X]_\varrho \to [\textbf{done}]_\varrho$$

$$X \;=\; \oplus[\texttt{Leaf} : \textbf{done}, \texttt{Node} : !\alpha ; X ; X]$$

# outline

# assessment

- ☹ explicit resumptions in code
- ☺ resumptions are sparse and their locations easy to spot
  - recursive calls not in tail position
- ☺ off-the-shelf type checking and inference

# on portability

## Required ingredients

- parametric polymorphism
- inference engine

## Safety of resumptions

- statically guaranteed ⇒ existential types
- dynamically guaranteed ⇒ lightweight runtime check

```
let (@>) f u =
  let v = Obj.magic f u in
  if same_endpoint u v then v else raise Error
```

# related work on type-level identities

- Launchbury & Peyton Jones, **State in Haskell**, 1995
- Walker & Watkins, **On regions and linear types**, 2001
- Ahmed, Fluet, Morrisett, **$L^3$: A linear language with locations**, 2007
- Charguéraud & Pottier, **Functional translation of a calculus of capabilities**, 2008
- Tov & Pucella, **Practical affine types**, 2011

happy hacking with FuSe

`http://di.unito.it/luca`