Polymorphic Context-free Session Types

Vasco T. Vasconcelos [joint with Bernardo Almeida, Andreia Mordido & Peter Thiemann]

Dagstuhl, 13 September 2021

FreeST is

- A functional language with
- Support for multithreading (message passing and choice), equipped with
- Session types (context-free, recursion),
- Polymorphic
- Haskell-like syntax
- Call by value
- Impure

Releases

- FreeST 1.0 _ Jul 2020
    - Predicative polymorphism (based on ICFP'17)
- FreeST 2.0 _ Feb 2021
    - System F^{μ} (based on paper in arXiv, submitted)
    - Richer kinding system
- FreeST 3.0
    - (To be discussed later on)

Google: freest lasige

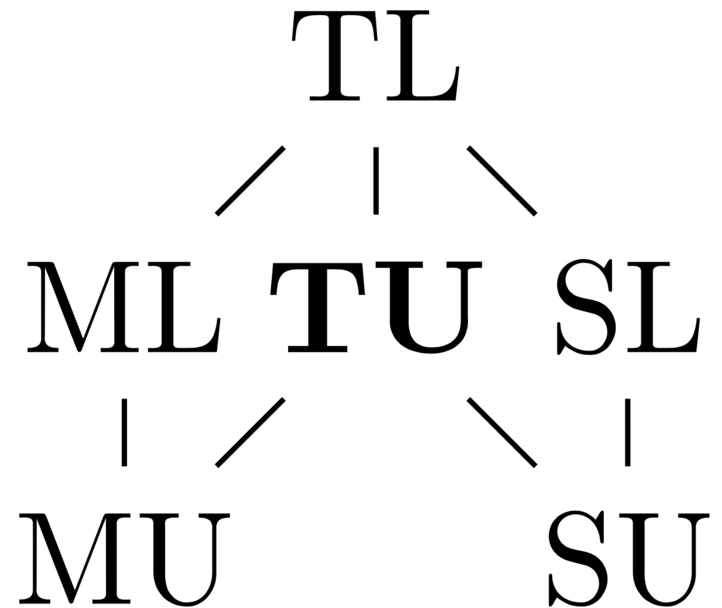
http://rss.di.fc.ul.pt/tools/freest/
http://rss.di.fc.ul.pt/tryit/FreeST

Variations on abstracted send and receive operations

Polymorphic lambda calculus

Kinds and Subkinding

- M _ Message
- S _ Session
- T _ Type (Top)
- L _ Linear
- U _ Unrestricted (Unlimited)
- TU _ default



Types for primitive operators

```
receive : ∀a:ML . ∀b:SL . ?a;b -> (a,b)
send    : ∀a:ML . a -> ∀b:SL . !a;b -o b
fork    : ∀a:TL . a -> ()
fst     : ∀a:TL . ∀b:TU . (a,b) -> a
snd     : ∀a:TU . ∀b:TL . (a,b) -> b
```

- No equally pleasing way to abstract over the choice (select and match) operators

Taking advantage of send as an operator of a rank 2 type

Context-free sessions

Type abstraction conflicts with linearity

A bit of the future

Duality and channel creation

$$\frac{\Delta \vdash \Gamma : \mathbf{T}^{\mathsf{un}} \qquad \varepsilon \vdash T : \mathbf{S}^{\mathsf{lin}}}{\Delta \mid \Gamma \vdash \mathsf{new}\, T : (T, \overline{T})} \;\textsc{T-New}$$

- Minimum support required: a function that builds a type dual to a given type (overline T)

The dualof operator is quite handy

FreeST 2.1

- Occurrences of the dualof operator disappear during the elaboration phase of the compiler.
- Before elaboration:

```
type T = !Int

f : dualof T -> Int
f c = fst [Int, Skip] $ receive c
```

- After elaboration (before type checking):

```
f : ?Int -> Int
```

Dualof of a polymorphic variable

FreeST 2.1

- Problem: polymorphic variables

```
newThunk : ∀a:SL . () -> (a, dualof a)
newThunk = Λa:SL => λ_ -> new a
```

- How do we get rid of dualof when applied to a polymorphic type variable?

```
f.fst:1:30: error:
         Cannot compute the dual of a polymorphic variable: a
```

Dualof for polymorphic types

FreeST 2.2

- Introduce co-variables (Lindley Morris, ICFP 16)
- Treat the dualof operator as in the De Morgan Laws (cf. treatment of negation in Linear Logic), getting rid of all occurrences of dualof except those applied to a type variable; these become co-vars
- Extend type equivalence to account for co-variables

$$\textsc{L-Var} \quad a \xrightarrow{a} \mathsf{Skip} \qquad \textsc{L-CoVar} \quad \overline{a} \xrightarrow{\overline{a}} \mathsf{Skip}$$

The run abstraction

Freest 2.2

- New becomes a conventional (primitive) poly function
- And we can abstract a quite common pattern: channel creation together with fork (cf. LL interpretations of session types)

```
run : ∀a:SL b:TU c:TL . (a -> b) -o (dualof a -> c) -o c
run f g =
  let (x, y) = new [a] in
  fork $ f x;
  g y
```

Higher-order polymorphism, Fω

FreeST 3.0

- Introduce arbitrary type operators
- Then we could have Dualof as a conventional type operator

```
List   : TU => TU
Dualof : SL => SL
```

Further extensions

- Pattern-matching for function definition
- Shared state, shared channels and races
- Inference of type application
- …
- (Polymorphism on lambda or on sessions?)