**FREEST** \_ CONTEXT-FREE SESSION TYPES IN CONCURRENT FUNCTIONAL LANGUAGE

**Vasco T. Vasconcelos**, Bernado Almeida, Andreia Mordido University of Lisbon

Programming Languages for Distributed Systems Shonan, May 2019

ISIGE

driven by excellence

Cheese

I milk

Myogurt

LASIGE

driven by excellence

Mcheese I milk

Myogurt

LASIGE

driven by excellence

***hee*se** M milk

Myogurt

*chees*e

LASIGE

driven by excellence

***hee*se** M milk

Ayogurt

*chees*e /

LASIGE

driven by excellence

***hee*se** M milk

Ayogurt

*cheese*

***milk***

sogurt

LASIGE

driven by excellence

***hee*se** M milk

Ayogurt

*cheese*

**m*ilk***

pos.Por

y sogurt

LASIGE

driven by excellence

*ch*eese

**UT**

sogurt

REGULAR SESSION-TYPES

type ListServer = &{

Nil : end,

Cons: ?Int . ListServer } The type of the server Choice Choice label

Termination

Receive Recursion

LASIGE

driven by excellence

LASIGE

driven by excellence

LASIGE

driven by excellence

LASIGE

driven by excellence

LASIGE

driven by excellence

LASIGE

driven by excellence

LASIGE

driven by excellence

LASIGE

driven by excellence

LASIGE

driven by excellence

ལས་ཀ་གོ་

LASIGE

driven by excellence

1

SESSION TYPES FOR LIST AND TREE SERVERS

type ListServer = &{

Nil: End, Cons: ?Int .

ListServer }

type TreeServer = &{

LeafS: Skip, NodeS: TreeServer ; ?Int ; TreeServer }

TYPE EQUIVALENCE \_ SOME LAWS

• Recursion is silently unfolded (equirecursive)

• Identity laws

• Associativity

• Distributivity

T ; skip ∼ T ; skip ∼ T

(T ; U) ; V ∼ T ; (U ; V)

(&{l:T, m:U});V) ∼ &{l:T;V, m:U;V}

**FREEST** CONTEXT-FREE SESSION TYPES IN A CONCURRENT FUNCTIONAL LANGUAGE

DEMO AVAILABLE

LASIGE

driven by excellence

tree Sum : forall a => dualof TreeC;a -> (Int, a) treeSum c =

match c with {

LeafC C -> (0, c); NodeC c ->

let x, c = receive c in let 1, c = treeSum[dualof TreeC;!Int;a] c in let r, c = treeSum[!Int;a] c in (x + 1 + r, send c (x + 1 + r))

main : Tree main =

let w, r = new TreeC in let - = fork (treeSum[Skip] r) in fst (transform[Skip] aTree w)

RAISING A TREE

data Tree =

Leaf | Node Tree Int Tree

raiseTree c =

case c of

LeafS c0 → (Leaf, c0)

choice label

NodeS c0 →

let (t1, c1) = raiseTree c0

(v, c2) = receive c1 (t2, c3) = raiseTree c2 in (Node t1 v t2, c3)

AIM: LOW-LEVEL SOLUTION

• Communication restricted to base types (no structures, no closures)

• First-order session types (no channel passing)

RAISING A LIST

type ListServer = &{

Nil : end, Cons: ?Int . ListServer }:: ListServer

raiseList c =

branch on a choice type

case c of

Nil → λc0. ([], c0) Cons → λc0.

let (x, c1) = receive c0

(xs, c2) = raiseList c1 in (x:xs, c2)read from a

channel

WHAT TYPING SHOULD GUARANTEE

• No ill-formed streams

Tree Int Tree

Node 5 Leaf Leaf

• Stream does not end too soon

Node Leaf 5

missing right tree

• No extra elements after the end of the stream

Leaf 5

type ListServer = &{

Nil : end, Cons: ?Int . ListServer }

• Regular trace language

(&Cons . ?Int)∗ . &Nil

RECALL THE LIST SERVER

THE TRACE LANGUAGE OF A SESSION TYPE

• The trace language of any (first-order) session type is:

• recognised by a finite automaton

• a (ω-) regular language

• The trace language of raiseTree is a context-free language: T → &LeafS

T → &NodeS T ?Int T

THEREFORE...

• There is no standard session type for raiseTree . It cannot be typed in (e.g.) GV

ENTER CONTEXT-FREE SESSION TYPES

REGULAR & CONTEXT-FREE SESSION TYPES

Regular Context free

!B.S !B

?B.S ?B end S;S skip

Hardwired sequencing of

communication Explicit sequencing

Tail recursive Unrestricted recursion

ARITHMETIC EXPRESSION SERVER type TermChan = &{Const: !Int,

Add: TermChan;TermChan, Mult: TermChan;TermChan} computeService :: TermChan;!Int —o skip computeService c =

let n,c1 = receiveEval c in send n c1 receiveEval :: ∀α.TermChan;α —o Int ⊗ α receiveEval c =

case c of {

Const c1 → receive c1 Add c1 → let n1,c2 = receiveEval 1 n2,c3 = receiveEval c2 in (n1+n2,c3) Mult c1 → let n1,c2 = receiveEval c ... }

REQUIREMENTS FOR THE META THEORY

TYPING RAISETREE

type TreeServer = &{

LeafS : skip,

• At the top level:

NodeS : TreeServer ;

?int ; TreeServer}

Linear function Linear pair

raiseTree :: TreeServer —o (Tree ⊗ skip)

TYPING RAISETREE

type TreeServer = &{

• At the first recursive call

LeafS : skip, NodeS : TreeServer ;

?int ; raiseTree c = case c of

TreeServer} Node → λc0. let (t1 , c1) = raiseTree c0

(v, c2) = receive c1 (t2 , c3) = raiseTree c2

raiseTree :: (TreeServer;?Int;TreeServer ;...) —o

Tree ⊗ (?Int; TreeServer;...)

TYPING RAISETREE

• At the top level:

raiseTree :: TreeServer —o (Tree ⊗ skip)

• At the first recursive call

raiseTree :: (TreeServer;?Int;TreeServer ;...) —o

Tree ⊗ (?Int; TreeServer;...)

• A polymorphic type

raiseTree :: ∀α. TreeServer;α —o (Tree ⊗ α)

INSTANTIATING A POLYMORPHIC TYPE

raiseTree :: ∀α. TreeServer;α —o (Tree ⊗ α)

• Replace α with skip

raiseTree :: TreeServer; skip —o (Tree ⊗ skip)

• We need

TreeServer; skip ~ TreeServer

TYPE EQUIVALENCE IS A BISIMULATION

• ... over a suitable labelled transition system

• How do we decide type equivalence?

• For regular types there is a simple (polynomial) algorithm

• But context-free types is a different story...

TYPE EQUIVALENCE IS DECIDABLE

• Via translation to BPA, Basic Process Algebra [Bergstra, Klop 1988]

• Bisimulation for BPA is decidable [Christiansen, Hüttel, Stirling 1995]

• Unfortunately, the decidability result does not yield an obvious algorithm

CONCLUSION

• Programming with context free session types is fun

• Compiler ready soon

• Implementing type equivalence challenging