Programming with $T \mathbin{;} U$

Bernardo Almeida, Diogo Barros, Diana Costa, Alcides Fonseca, Guilherme Lopes, Paula Lopes, Andreia Mordido, Diogo Poças, Miguel Roldão, João Roque, Afonso Rosa, Gil Silva, António Silvestre, Peter Thiemann, Vasco T. Vasconcelos,
University of Lisbon, University of Freiburg

PL@LX, Técnico, Lisbon, June 3, 2024

back to the classics

Types for Dyadic Interaction

Kohei Honda

kohei@mt.cs.keio.ac.jp

Department of Computer Science, Keio University

3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We formulate a typed formalism for concurrency where types denote freely composable structure of dyadic interaction in the symmetric scheme. The resulting calculus is a typed reconstruction of name passing process calculi. Systems with both the explicit and implicit typing disciplines, where types form a simple hierarchy of types, are presented, which are proved to be in accordance with each other. A typed variant of bisimilarity is formulated and it is shown that typed β-equality has a clean embedding in the bisimilarity. Name reference structure induced by the simple hierarchy of types is studied, which fully characterises the typable terms in the set of untyped terms. It turns out that the name reference structure results in the deadlock-free property for a subset of terms with a certain regular structure, showing behavioural significance of the simple type discipline.

Concur 1993

the elements of dyadic interaction

?T

(input a value)

!T

(output a value)

Wait	Close
(wait for close)	(close channel)

$\forall a.T$	$\exists a.T$
(input a type)	(output a type)

$T \& U$

(offer choice of T or U)

$T \oplus U$

(select from T or U)

$\&\{l_1: T_1, \ldots, l_n: T_n\}$

(labelled offer)

$\oplus\{l_1: T_1, \ldots, l_n: T_n\}$

(labelled selection)

T ; U	Skip
(sequential composition)	(identity)

$\mu a.T$	a
(recursion)	(self reference)

duality

Dual (?T)	!T
Dual (!T)	?T
Dual Wait	Close
Dual Close	Wait
Dual (&{l: T, m: U})	⊕{l: Dual T, m: Dual U}
Dual (⊕{l: T, m: U})	&{l: Dual T, m: Dual U}
Dual (T ; U)	Dual T ; Dual U
Dual Skip	Skip

the freest programming language

- Functional (system F)
- Concurrent
- Call-by-value
- Message-passing on bidirectional, heterogeneous channels
- Buffered channels (asynchronous message passing)
- Linear and shared (unrestricted) channels
- Channel behaviour (protocol) described by types

infinite streams

```
type IRepeat a = a ; IRepeat a

type IStream a = IRepeat (!a)          -- seen from the writer
type CoIStream a = Dual (IStream a) -- seen from the reader
--                  = IRepeat (?a)

-- A consumer for type CoIStream a
echo : CoIStream a -> Diverge
echo c =
  let (x, c) = receive c in print x ; echo c

-- A consumer for type IStream Int
ints : Int -> IStream Int -> Diverge
ints n c = ints (n + 1) (send n c)
         = let c = send n c in ints (n + 1) c -- alternative
         = c |> send n |> ints (n + 1)        -- preferred
```

running

```
echo : Dual (IStream a) -> Diverge

ints : Int -> IStream Int -> Diverge

main : Diverge
main =
  forkWith (ints 0)
  |>
  echo
```

```
$ freest iStream.fst
0
1
2
3
...
```

more infinite streams

```
type Serve a b = IRepeat (!a ; ?b)

-- A consumer for type Dual (Serve Int Bool)
gzServer : Dual (Serve Int Bool) -> Diverge
gzServer c =
  let (n, c) = receive c in
  c |> send (n > 0) |> gzServer
```

finite and infinite streams

```
type Repeat a = +{ More: a ; Stream, Done: Skip }

type Stream a = Repeat (!a)

foldl : ?(b -> a -> b) ; ?b ; Dual (Stream a) ; !b ; Wait -> ()
foldl r =
  let (f, r) = receive r in
  let (e, r) = receive r in
  let (x, r) = fldl f e r in
  r |> send x
    |> wait
  where
    fldl : (b -> a -> b) -> b -> Dual (Stream a) ; c -> (b, c)
    fldl _ x (Done r) = (x, r)
    fldl f x (More r) = let (y, r) = receive r in fldl f (f x y) r
```

sharing

interference

```
f : () -> ()
f _ =
  print "Hello " ;
  print "FreeST"

main : ()
main =
  fork f ;
  f ()
```

```
$ freest io.fst
Hello
Hello
FreeST
FreeST
```

*?T

~ ?T ; ?T ; ...

(forever input a value)

*!T

~ !T ; !T ; ...

(forever output a value)

$*\&\{l_1, \ldots, l_n\}$

$\sim \&\{l_1 : \&\{l_1 : \ldots\} \ldots, l_n : \&\{l_1 : \ldots\}\}$

(forever offer a set of labels)

$*\oplus\{l_1, \ldots, l_n\}$

$\sim \oplus\{l_1 : \oplus\{l_1 : \ldots\} \ldots, l_n : \oplus\{l_1 : \ldots\}\}$

(forever select from a set)

stdout is of a star type

```
type StdOutSession = +{ PutChar : !Char ; StdOutSession
                      , PutStr   : !String ; StdOutSession
                      , PutStrLn: !String ; StdOutSession
                      , Done     : Close
                      }

type StdOut : *?StdOutSession

stdout : StdOut

-- Receive a value from a star channel
-- Session initiation on the reader side
receive_ : *?a -> a
receive_ c = c |> receive |> fst
```

controlling interference

```
g : () -> ()
g _ =
      receive_ stdout
   |> select PutStr    |> send "Hello "
   |> select PutStrLn  |> send "FreeST"
   |> select Close     |> close

main : ()
main =
  fork g ;
  g ()
```

```
$ freest io.fst
Hello FreeST
Hello FreeST
```

https://freest-lang.github.io/

freest 3.2 in numbers

First git commit	20/11/2017
Commits	3557
Contributors	12
LOC (Haskell, Happy, Alex, Freest)	5741
Manual tests	1046
LOC (manual tests)	9773
Quickcheck for	Type equivalence
Support for	Studio Code, Emacs
Runs on	Linux, MacOS, Windows
Theses	1 PhD (ongoing), 8+ MsC

freest biography

First-order messages, Damas-Milner type system	Nov 2017	1.0.0
Impredicative polymorphism (System F)	Fev 2021	2.0.0
Higher-order messages	Apr 2023	3.0.0
Pattern matching	Apr 2023	3.0.0
Shared channels	Apr 2023	3.0.0
Channel closing	Nov 2023	3.1.0
Kind inference	Apr 2024	3.2.0
Subtyping	2024	
Local type inference	2024	
Send/receive types	2024/2025	
Type operators	2024/2025	

the technical challenge ...

… is type equivalence

type equivalence

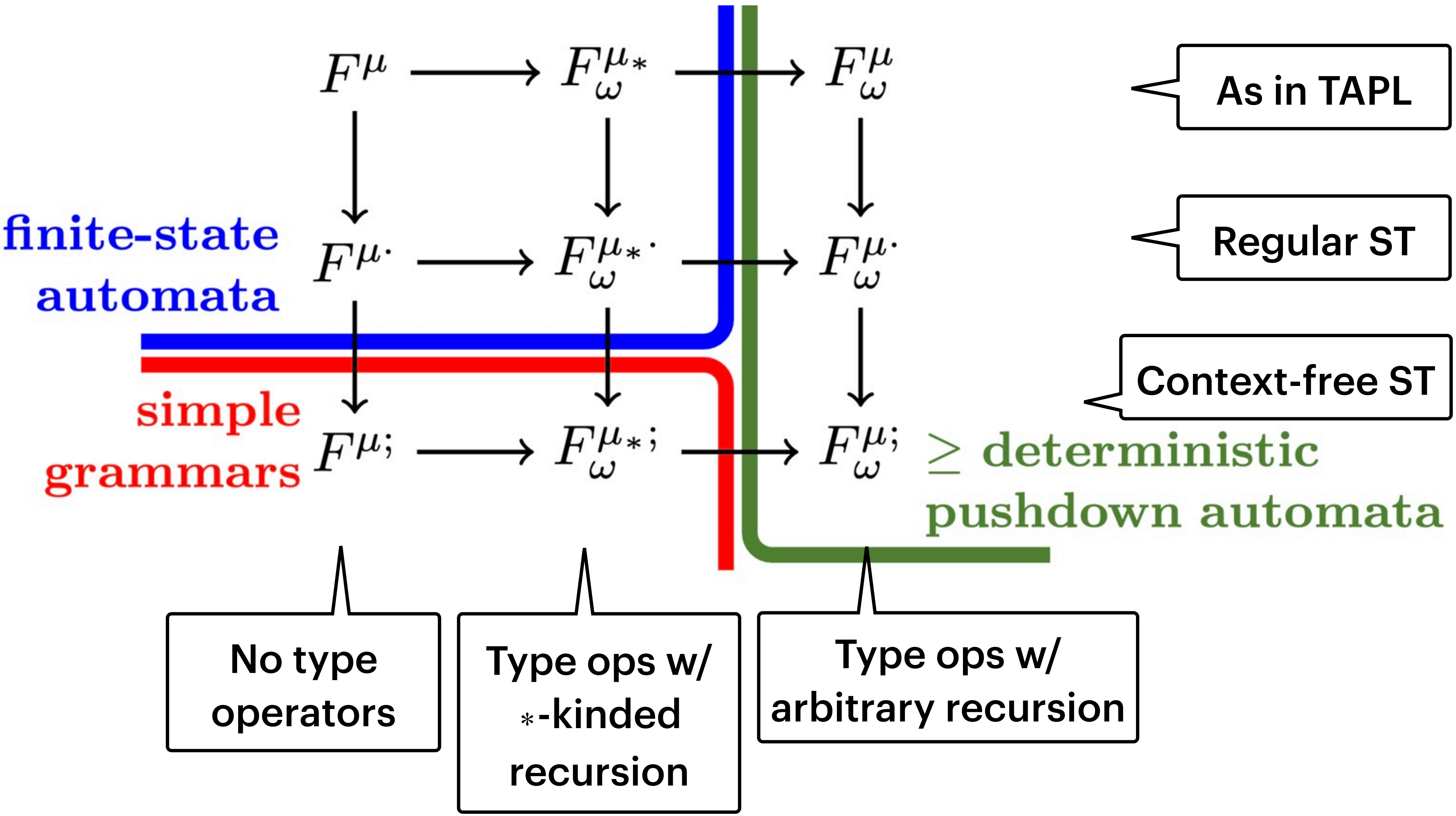
- Determined by a bisimulation game between two types, T and U:
  - T must simulate U
  - U must simulate T
- Or else by a coinductive system of rules

laws of sequential composition

(T ; U) ; V ~ T ; (U; V)	Associativity
T ; Skip ~ Skip ; T ~ T	Skip is identity
+{l: T, m: U} ; V ~ +{l: T ; V, m: U ; V}	Right distributivity
Close ; T ~ Close	Close is left zero (same for Wait)
(μa. T ; a) ; U ~ μa. T ; a	Unnormed types are left absorbing

deciding type equivalence

1. Transform the two types into two words of a simple grammar
   - Simple grammars are deterministic context-free grammars in Greibach normal form
2. Run a bisimulation algorithm on simple grammars
   - Currently: a doubly-exponential algorithm
   - Coming soon: single-exponential algorithm



F^{μ}
$F^{\mu *}_{\omega}$
F^{μ}_{ω}
$F^{\mu\cdot}$
$F^{\mu *\cdot}_{\omega}$
$F^{\mu\cdot}_{\omega}$
$F^{\mu;}$
$F^{\mu *;}_{\omega}$
$F^{\mu;}_{\omega}$
As in TAPL
Regular ST
Context-free ST
finite-state automata
simple grammars
$\geq$ deterministic pushdown automata
No type operators
Type ops w/ $*$-kinded recursion
Type ops w/ arbitrary recursion

freest 5.0

what shall be FreeST 5.0?

- A complete reimplementation, incorporating
  - A new AST much closer to the source language, better suited for inference, error message issuing, and back-ends
  - Type operators
  - Support for receive/send types
  - A new algorithm to decide bisimulation

would like to contribute?

- We have just started implementing 5.0
- There is a lot to do
  - Language design
  - Implementation
  - Developing in FreeST
- We are planning a paper on a major conference

coming soon to a bookstore near you

https://freest-lang.github.io/