FreeST and the Higher-order Polymorphic Lambda Calculus

Bernardo Almeida, Diana Costa, Andreia Mordido,
Diogo Poças, **Vasco T. Vasconcelos**

University of Porto, 6 October 2022

FreeST is …

A programming language

- Functional
- Concurrent
- Call-by-value
- Message-passing on bidirectional, heterogeneous channels
- Linear and shared (unrestricted) channels
- Channel behaviour (protocol) described by types
- Types: Polymorphic (System F), recursive, context-free session types

FreeST in numbers

- First git commit: 20/11/2017
- 8 git contributors
- 3270 commits into the dev branch alone
- 4392 LOC (Haskell, Happy, Alex, FreeST)
- 817 manual tests (6549 FreeST LOC)
- 150135 quick check tests (type equivalence)
- Support for Visual Studio Code, Atom, Emacs
- Runs on Linux, MacOS, Windows
- 1 PhD thesis (ongoing)
- 4 + 2 MSc thesis (completed + ongoing)

FreeST in Action

Lists

- Lists are the bread and butter of functional programming
- Yet FreeST features no primitive support for lists
- One may write

```
data IntList = ICons Int IntList | INil

data BoolList = BCons Bool BoolList | BNil
```

- But not

```
data List a = Cons a (List a) | Nil
```

What's so difficult about polymorphic lists, anyway?

```
data List a = Cons a (List a) | Nil
```

- List is not a type as we know them, but a type operator
- When applied to Int, as in List Int, it becomes a proper type
- In any case, the theory of Higher-order Polymorphism, F_ω, is well established
- Why are we taking so long?

The answer is “the semicolon is holding us”

Type equivalence is a bisimulation

- But first let us understand how we decide type equivalence
- <Whiteboard here>

Deciding type equivalence

- Rather than looking for fixed-point as just shown, we
- Translate types into simple grammars:
  - Productions of the form $X \longrightarrow a\ X_1 \ldots X_n$ ($n >= 0$)
  - No epsilon transitions
  - Productions are deterministic: no
    - $X \longrightarrow a\ Y_1 \ldots Y_n$ and
    - $X \longrightarrow a\ Z_1 \ldots Z_m$
- Bisimulation for simple grammars is decidable; there is a practical algorithm

Higher-order Polymorphism in FreeST

- First-order

$$\mathsf{IntStream} = \mu\,\alpha : \mathrm{S}.\,\&\{\mathsf{Done} \colon \mathsf{End}, \mathsf{More} \colon ?\mathsf{Int}; \alpha\}$$

- Higher-order

$$\mathsf{Stream} = \lambda\alpha : \mathrm{T}.(\mu\,\beta : \mathrm{S}.\,\&\{\mathsf{Done} \colon \mathsf{End}, \mathsf{More} \colon ?\alpha; \beta)\}$$

- Is IntStream equivalent to Stream Int?
- We need beta-reduction at the type level

$$(\lambda\alpha : \kappa.T)\ U \longrightarrow_\beta T[\alpha \mapsto U]$$

- But simple grammars don't know how to beta-reduce :(

The type-level Dual operator can be internalised

- We have seen the dualof macro
- We can now write

```
streamify : ∀a.∀c.∀d.TreeC a; c → Dual (Stream a) ; d
                                 → (c, Dual (Stream a); d)
```

- where Dual is an operator of kind S —> S (from session types to session types)

The expressive power of extensions to System F

F _ Polymorphic lambda-calculus

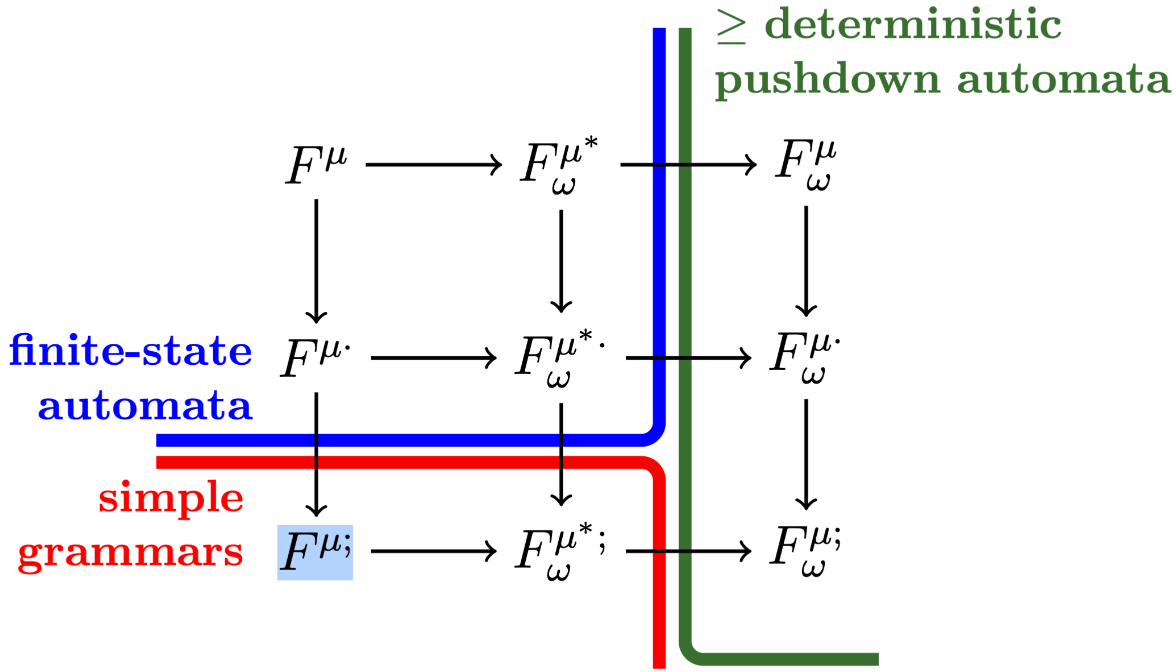
F^{μ} _ F with (equi) recursive types

$F^{\mu*}$ _ F w/ monomorphic recursion

$F^{\cdot}$ _ F with tail-recursive ST

$F^{;}$ _ F with context-free ST

F_{ω} _ Higher-order polymorphism



Expressive power
(arrows denote strict inclusions)

System F_ω^μ with Context-free Session Types

$*$ ::=		Base kind
	S	Session
	T	Functional
κ ::=		Kind
	$*$	kind of types
	$\kappa \Rightarrow \kappa$	kind of type constructors
T ::=		Type or type constructor
	ι	type constant
	α	type variable
	$\lambda\alpha\colon \kappa.T$	type-level abstraction
	$T\,T$	type-level application

Fig. 3: The syntax of types

ι ::=			Type constant
	Skip	S	skip
	End	S	end
	$\sharp$	$* \Rightarrow \mathrm{S}$	input and output
	$;$	$\mathrm{S} \Rightarrow \mathrm{S} \Rightarrow \mathrm{S}$	sequential composition
	$\odot_{\{\overline{l_i}\}}$	$\overline{\mathrm{S} \Rightarrow} \mathrm{S}$	external and internal choice
	$\rightarrow$	$* \Rightarrow * \Rightarrow \mathrm{T}$	arrow
	$\forall_\kappa$	$(\kappa \Rightarrow *) \Rightarrow \mathrm{T}$	universal type
	Unit	T	unit
	$(\!|\overline{l_i}|\!)$	$\overline{* \Rightarrow} \mathrm{T}$	record and variant
	μ_κ	$(\kappa \Rightarrow \kappa) \Rightarrow \kappa$	recursive type
	Dual	$\mathrm{S} \Rightarrow \mathrm{S}$	dual type constructor

Fig. 4: Type constants and their kinds

The labelled-transition system

- Some rules

$$!T;U \xrightarrow{!_1} T \qquad !T;U \xrightarrow{!_2} U$$

$$\lambda\alpha\colon \kappa.T \xrightarrow{\lambda\alpha\colon\kappa} T$$

$$\frac{T \longrightarrow_\beta U \qquad U \xrightarrow{a} V}{T \xrightarrow{a} V}$$

- How do we check this goal

$$\lambda\alpha\colon \kappa.\alpha \text{ equivalent to } \lambda\beta\colon \kappa.\beta$$

if α and β, both bound variables, appear in the LTS as labels?

Minimal Renaming

- We take the set of type variables as ordered and
- Perform **minimal renaming** on all bound variables
- Example where v_1 is the first *non* free variable in each subterm

$$\mathrm{rename}(\lambda\alpha : \mathrm{T}.\lambda\beta : \mathrm{S}.\beta) = \lambda v_1 : \mathrm{T}.\lambda v_1 : \mathrm{S}.v_1$$

- And we also do this in beta-reduction because renaming is not preserved by reduction

$$(\lambda\alpha : \kappa.T)\, U \longrightarrow_\beta \mathrm{rename}_\emptyset(T[\alpha \mapsto U])$$

Deciding type equivalence

- Take the polymorphic tree receive type

```
type TreeC a = &{LeafC: Skip, NodeC: TreeC a; ?a ; TreeC a}
```

- which can be written as

$$T_0 = \lambda\alpha : \mathtt{T}.\mu\,\beta : \mathtt{S}.\,\&\{\mathsf{Leaf} : \mathsf{Skip}, \mathsf{Node} : \beta; ?\alpha; \beta\}.$$

- Translate to a simple grammar

$$X_0 \xrightarrow{\lambda v_1 : \mathtt{T}} X_1 \qquad X_1 \xrightarrow{\&_1} \varepsilon \qquad X_1 \xrightarrow{\&_2} X_3 \qquad X_2 \xrightarrow{\&_1} \varepsilon \qquad X_2 \xrightarrow{\&_2} X_3$$

$$X_3 \xrightarrow{\&_1} X_4 X_1 \qquad X_3 \xrightarrow{\&_2} X_3 X_4 X_1 \qquad X_4 \xrightarrow{?_1} X_5 \bot \qquad X_4 \xrightarrow{?_2} \varepsilon \qquad X_5 \xrightarrow{v_1} \varepsilon$$

- Do this to both types; run the bisim algorithm on the grammar

The current FreeST compiler

- The AST contains types in AST form
- Whenever we need to check type equivalence we
  - Convert both types to a grammar
  - Run bisimulation on the the grammar
  - Discard the grammar

The next FreeST compiler

- At the elaboration stage (between parsing and type checking) we translate all types to (words of) non-terminal symbols in a single grammar
- Rather than types in AST format we keep types as words of non-terminal symbols
- No need for to-grammar translation at type equivalence checking points
- Furthermore, extracting the main type operator in a type becomes a lot simpler. Here'a an algorithmic typing rule in the current compiler

$$\textsc{TA-App}\quad \frac{\Delta \mid \Gamma_1 \vdash e_1 \Rightarrow\Downarrow T \to_m U \mid \Gamma_2 \qquad \Delta \mid \Gamma_2 \vdash e_2 : T \Rightarrow \Gamma_3}{\Delta \mid \Gamma_1 \vdash e_1\, e_2 \Rightarrow U \mid \Gamma_3}$$

Conclusion

- We had a lot of fun until now
- We plan to continue having fun for some time
- A lot remains to be done
  - Implement higher-order polymorphism
  - Local kind inference for type abstractions and recursive types

```
forall a:1S . TreeC ; a -> (Tree, a)
```

  - Local type inference for type applications

```
forkWith @TreeChannel @() (writeTree aTree)
```