FreeST and Polymorphic Higher-order Session Types

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

University of Nagoya, 30 November 2023

FreeST is …

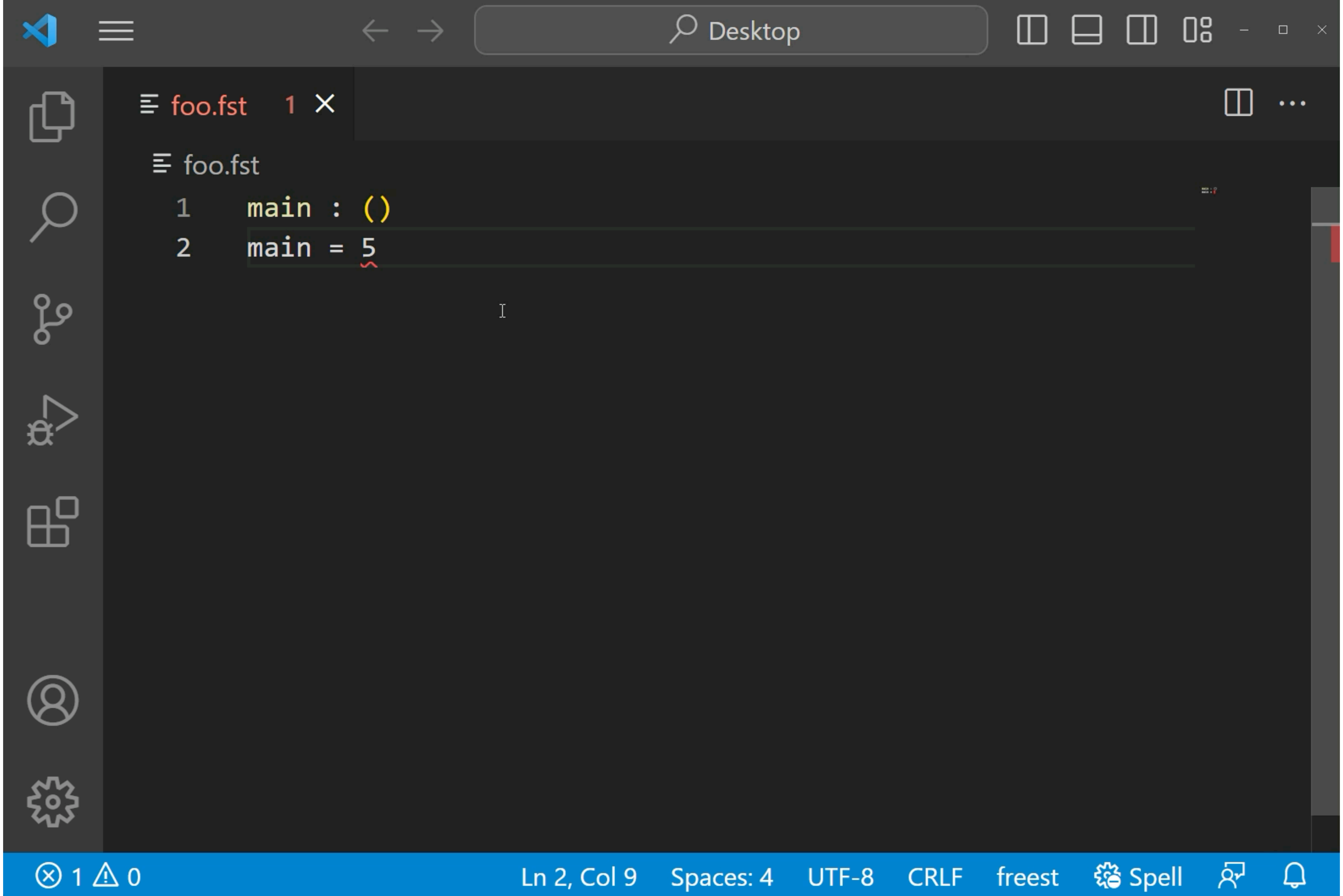
A programming language

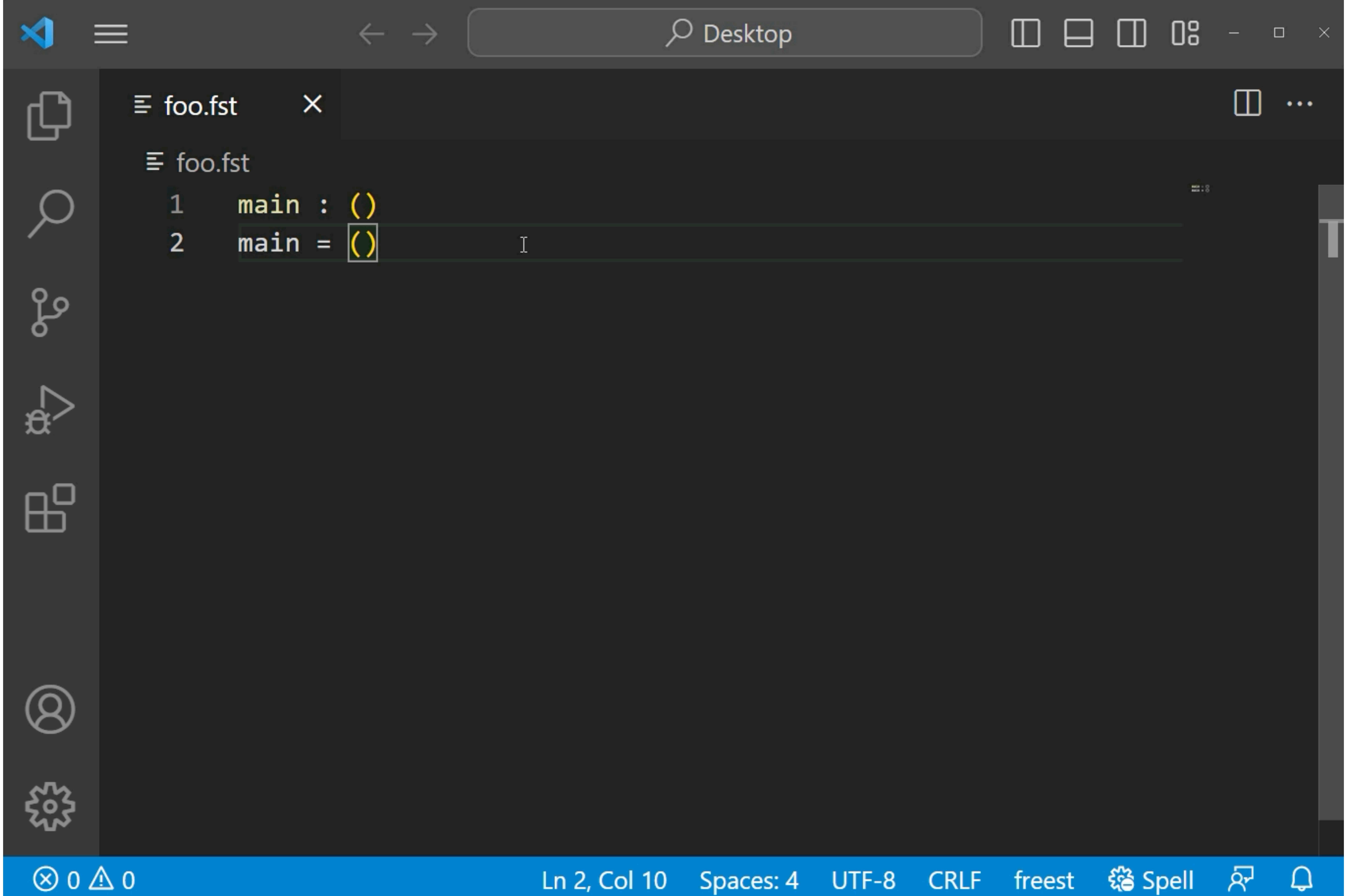
- Functional
- Concurrent
- Call-by-value
- Message-passing on bidirectional, heterogeneous channels
- Buffered channels (asynchronous message passing)
- Linear and shared (unrestricted) channels
- Channel behaviour (protocol) described by types
- Types: Polymorphic (unpredicative), recursive, higher-order context-free session types

FreeST in numbers

- First git commit: 20/11/2017
- 12 git contributors
- 3416 commits into the dev branch alone
- 4392 LOC (Haskell, Happy, Alex, FreeST)
- 977 manual tests (9749 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 + 4 MSc thesis (completed + ongoing)

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


Desktop
foo.fst 1
foo.fst
main : ()
main = 5
1 0
Ln 2, Col 9
Spaces: 4
UTF-8
CRLF
freest
Spell


Desktop
foo.fst
foo.fst
1 main : ()
2 main = ()
0 0
Ln 2, Col 10
Spaces: 4
UTF-8
CRLF
freest
Spell

FreeST in Action

(Demo time)

チャレンジ

The main challenge is type equivalence in the presence of semicolon

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 deductive coinductive system of rules (not shown)

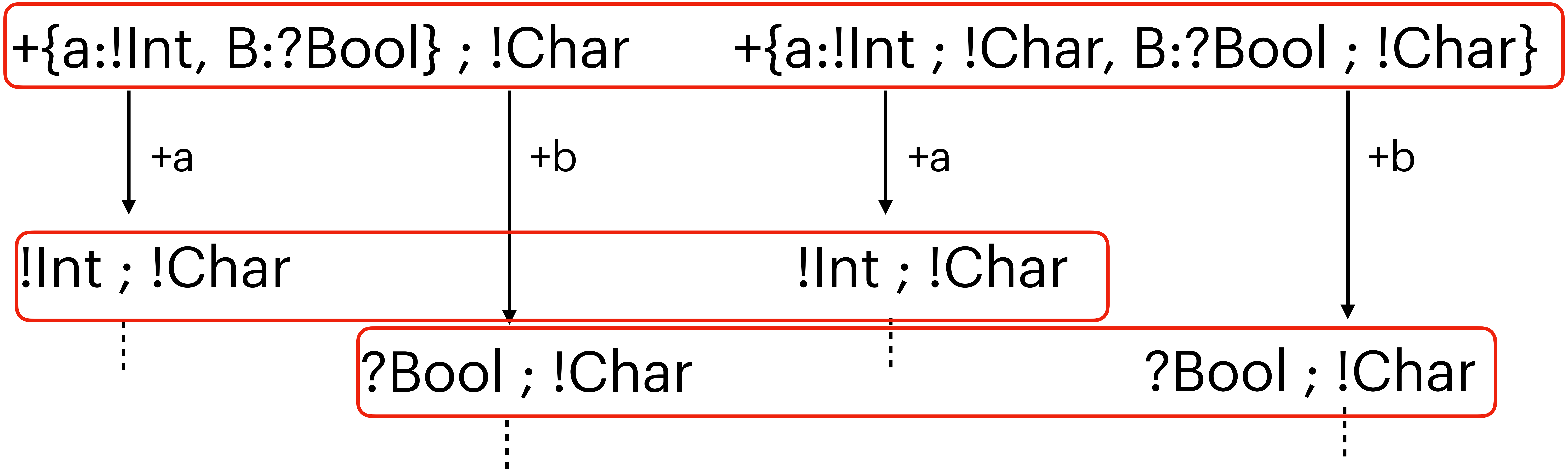
Some laws for sequential composition

(T ; U) ; V = T ; (U; V)	Associativity
T ; Skip = Skip ; T = T	Skip is neutral element
Close ; T = Close	Close is left absorbing (same for Wait)
+{a: T, b: U} ; V = +{a: T ; V, b: U ; V}	Right distributivity
(rec x. T ; x) ; U = rec x. T ; x	Unnormed types are left absorbing

Running a bisimulation on two types

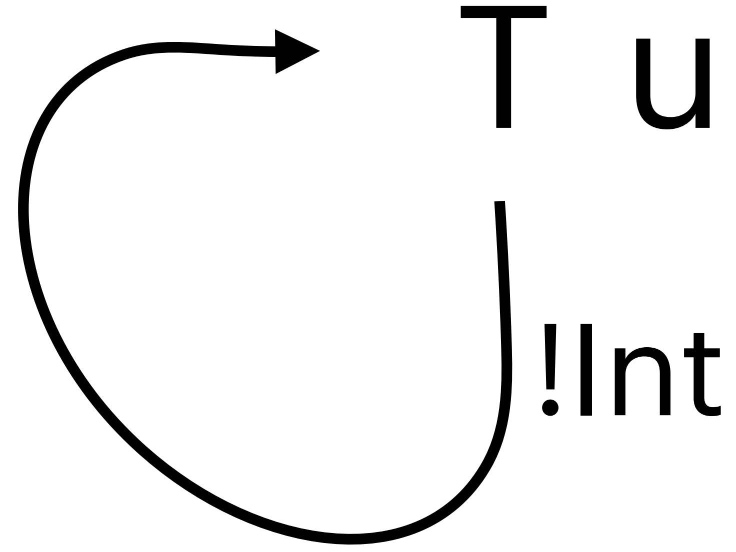
Simplified for first order sessions

Another example of bisimulation



Why not a standard fixed point construction?

Let T = rec x. !Int ; x



T unfolds to !Int; T

Let U = rec y. !Int ; y ; y

U unfolds to !Int; U ; U

↓ !Int

U ; U

↓ !Int

U ; U ; U

⋮

T and U are equivalent

The bisimulation is

$\{(T, U^n) \mid n \geq 1\}$

How do we decide type equivalence?

- Transform types into **simple grammars**
  - Productions of the form $X \rightarrow a\, Y_1 \ldots Y_n$, $n \geq 0$
  - No ε transitions
  - No two productions $X \rightarrow a\, Y_1 \ldots Y_n$ and $X \rightarrow a\, Z_1 \ldots Z_m$ (deterministic)
- We have developed an algorithm to decide the bisimilarity of two words in a grammar
- It is incorporated in the Freest compiler

The grammar associated to a type

- Type: +{a: !Int, b:?Bool} ; !Char
- Start word: X_1
- Productions:
  - $X_1 \rightarrow$ **+a** X_2X_4 $X_1 \rightarrow$ **+b** X_3X_4
  - $X_2 \rightarrow$ **!Int**
  - $X_3 \rightarrow$ **?Bool**
  - $X_4 \rightarrow$ **!Char**

A word in **bold** represents **one** terminal symbol

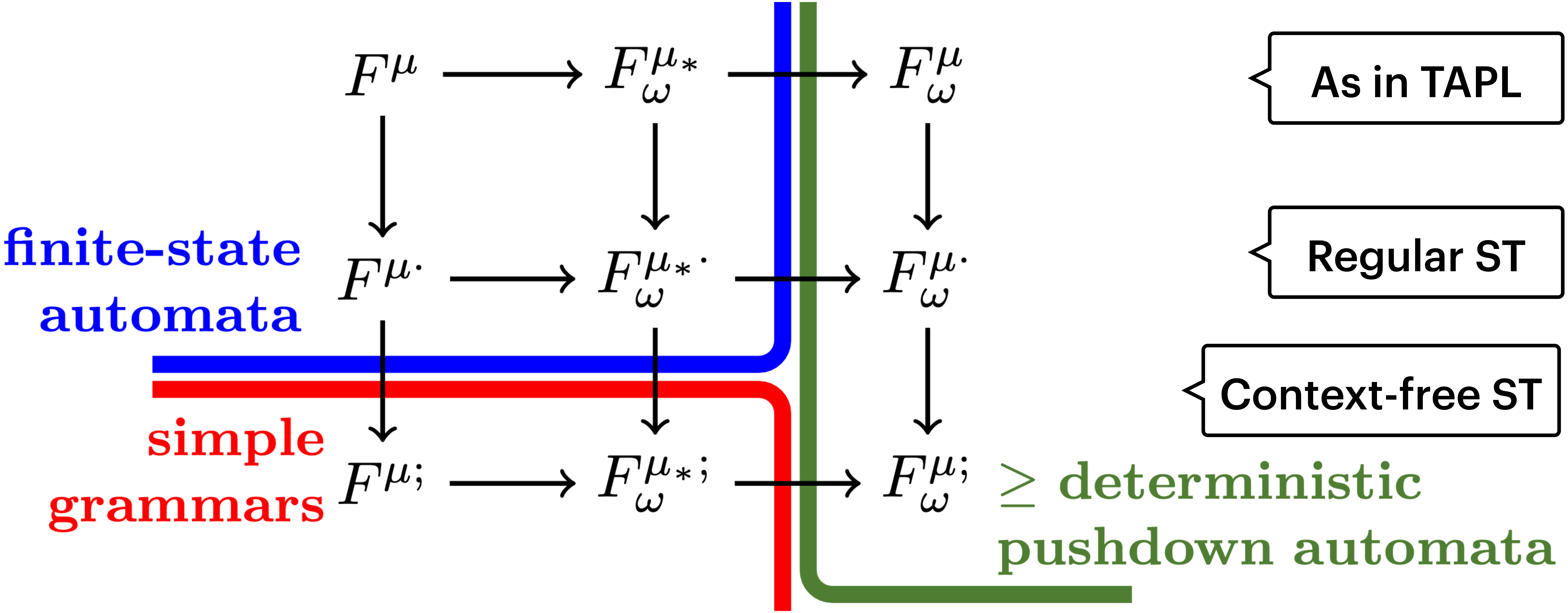
Which types can be translated to simple grammars?

- Predicative polymorphic + first-order session types (ICFP 2016)
- Impredicative polymorphic (System F^{μ}), still first order sessions (I&C 2022)
- System F^{μ} + higher-order session types (PLACES 2022, TCS soon)
- This is Freest V3.0
- Type operators (System $F^{\mu}{}_{\omega}$) with $*$-kinded recursion only, i.e., no recursion over type operators (ESOP 2023)
- (Are we reaching the limit?)

All systems include recursive types

First-order: channels carry base types only
Higher order: channels may carry channels

Where are the limits?



System $F^{\mu;}_{\omega}$ with Context-free Session Types

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

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

With type operators duality can be internalised

- We have seen the unmarshall function with the dualof macro

```
unmarshall : dualof TreeC ; a -> (Tree, a)
```

- We can now marshall and unmarshall trees of arbitrary types

```
unmarshall : dualof (TreeC b) ; a -> (Tree b, a)
```

- And we can have Dual as a type operator

```
unmarshall : Dual (TreeC b) ; a -> (Tree b, a)
```

- The Dual operator is of kind S → S (from session types to session types)

System F_ω^μ with Context-free Session Types

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

Only 4 types

Fig. 3: The syntax of types

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

Fig. 4: Type constants and their kinds

The labelled-transition system for type equivalence

- 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 different labels?

- Remember that labels in the LST are terminal symbols in grammars

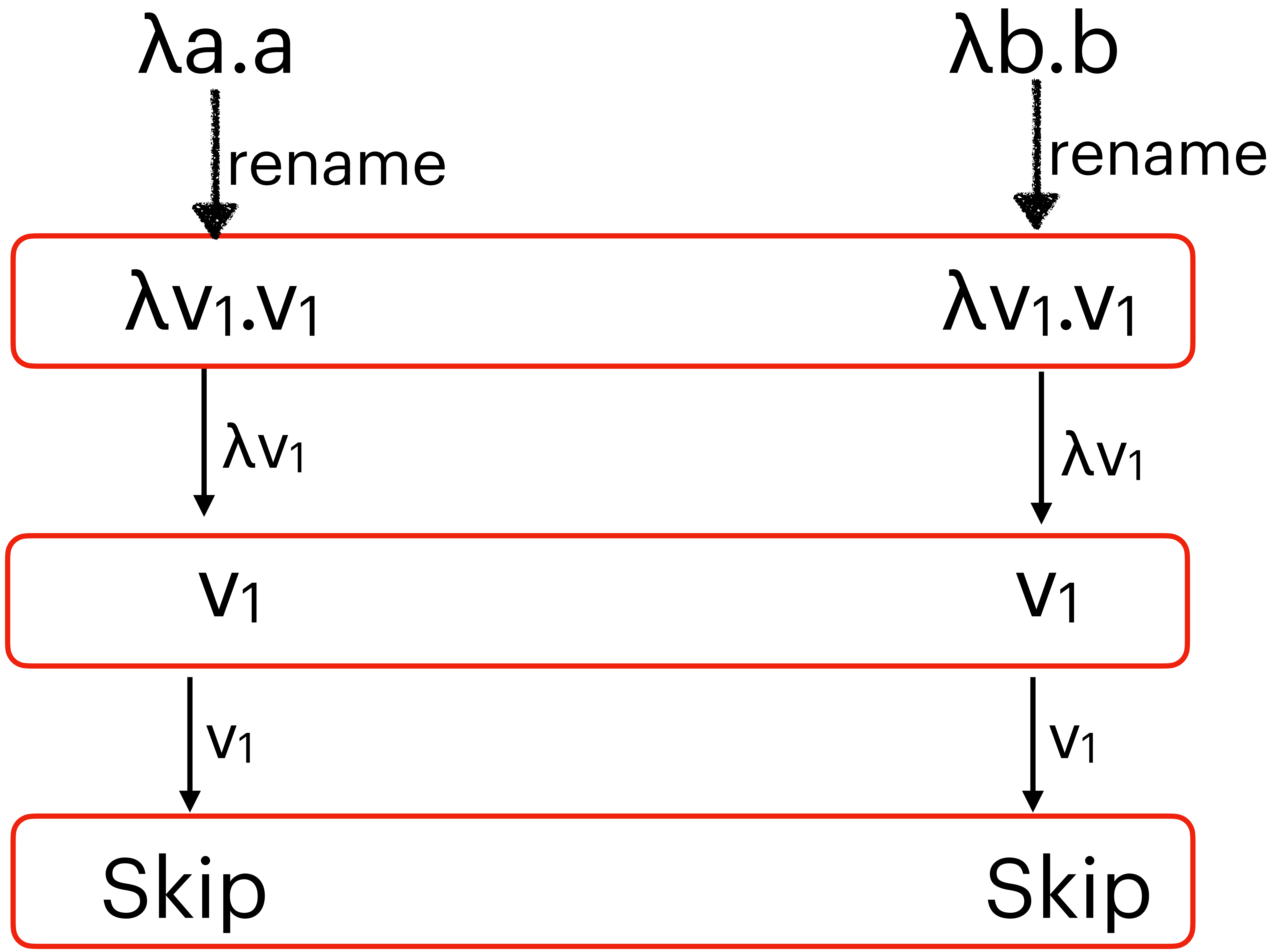
Solution: Minimal renaming

- 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

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

- We thus obtain types with variable names in canonical form and λ is not a binder anymore

Example


λa.a
λb.b
rename
rename
λv1.v1
λv1.v1
λv1
λv1
v1
v1
v1
v1
Skip
Skip

Back to FreeST

The current FreeST compiler

- The AST contains types in AST form
- Whenever we need to check type equivalence we
  - First check whether the two types are alpha-equivalent (linear)
  - If not:
    - Convert both types to a grammar
    - Run bisimulation on the the grammar
    - Discard the grammar (what a waste)

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 the types themselves, the AST keeps words of non-terminal symbols representing types
- 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's 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
  - Kind inference for type abstractions and recursive types (coming soon)

```
∀a:1S . Tree -> TreeC ; a -> a
```

  - Local type inference for type applications

```
forkWith @(dualof TreeC ; Wait) @() (marshallTree aTree)
```

  - Devise a faster algorithm for type equivalence (coming soon) λST

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