The types of dyadic interaction

A tribute to Kohei Honda

by Diana Costa, Andreia Mordido, Diogo Poças, and Vasco T. Vasconcelos
University of Lisbon

IFIP WG2.8 Viana do Castelo March 2026

How it all started

Types for Dyadic Interaction

Kohei Honda

kohei@mt.cs.keio.ac.jp

Department of Computer Science, Keio University

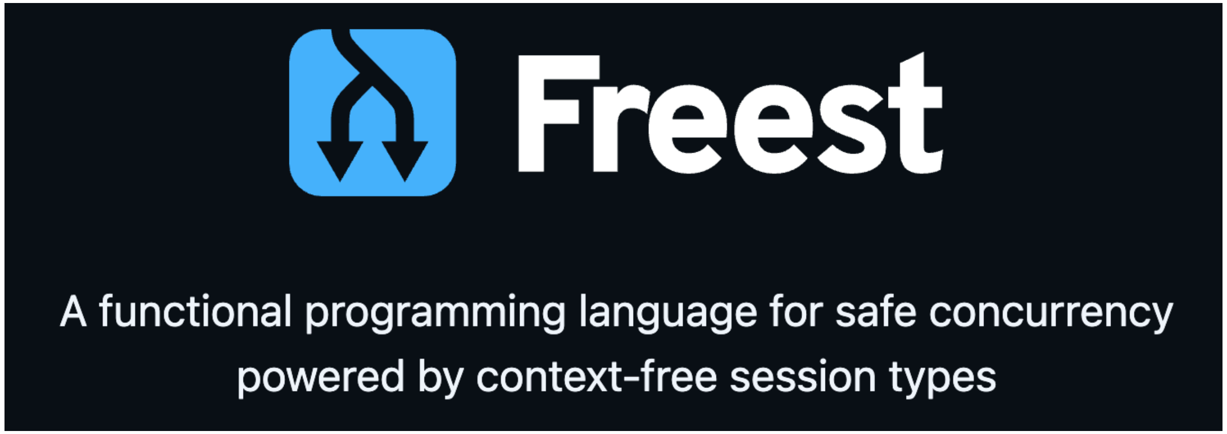
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

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 FreeST programming language



- Functional (call by value)
- Concurrent
- Communication by message passing (only)
- Channels are buffered, linear and governed by session types
- System $F^{\mu;}_{\omega}$

1 _ Programming dyadic interaction

The types of dyadic interaction

Blocking

Input

Output

Non blocking

Chain operations on values of these types

Pattern match on values of these types

Negative	**Positive**	
?	!	Value exchange
&	$\oplus$	Choice
$\forall$	$\exists$	Type exchange
Wait	Close	Channel closing

Infinitely repeating some action a

Multiplicity
(linear)

Kind
(Session)

```
type IRepeat : 1S -> 1S
type IRepeat a = a ; IRepeat a
```

Infinite streams of some type a

Multiplicity (unrestricted)

Kind (Type/any)

```
type IStream, CoIStream : *T -> 1S
type IStream a = IRepeat (?a)        -- seen from the reader
type CoIStream a = Dual (IStream a) -- seen from the writer
```

A (internalised) type operator

A consumer of type IStream Int

Pattern matching on negative type constructor

A type operator for non-inhabited types

```
echo : IStream Int -> Void @*T
echo (?x ; c) = printInt x ; echo c
```

Input x

Continue as c

```
type IStream Int ≃ ?Int ; ?Int ; ...
```

A consumer of type CoIStream

Reverse function application
x |> f = f x

Chaining operations on positive types

```
ints : Int -> CoIStream Int -> Void @*T
ints n c =
  c |> send n |> ints (n + 1)          -- preferred
  ints (n + 1) (send n c)              -- the functional way
  let c = send n c in ints (n + 1) c -- alternative
```

```
type CoIStream Int  ≃ !Int ; !Int ; ...
```

Finite or infinite streams

```
type IFRepeat : 1S -> 1S -> 1S
type IFRepeat a b = IRepeat (&{More: a, Done: b ; Wait})       -- unfold
--  ≃ &{More: a, Done: b ; Wait}; IFRepeat a b                 -- distributivity
--  ≃ &{More: a ; IFRepeat a b, Done: b ; Wait ; IFRepeat a b} -- Wait is absorbing
--  ≃ &{More: a ; IFRepeat a b, Done: b ; Wait}                -- fold
--  ≃ μc.&{More: a ; c, Done: b ; Wait}
```

A consumer of type IFRepeat (?a) Skip

```
length : IFRepeat (?a) Skip -> Int
length (&Done Wait) = 0
length (&More (?_ ; c)) = 1 + length c
```

Nested pattern matching on negative type constructors

```
type IFRepeat (?a) Skip ≃ μc.&{More: ?a ; c, Done: Wait}
```

Folding a stream of heterogeneous values

Type of the fold

Type of the element next element on the stream

The result of the fold operation

```
type Fold : 1S
type Fold = ∀(a:*T) . ?a ; IFRepeat (∀(b:1S) . ?(a -> b -> a) ; ?b) (!a)
```

Initial element

Function to fold next element

The next element

A consumer for type Dual Fold

Expect return value
“5True”

```
showStream : Dual Fold -> String
showStream c =
  let (x, c) = c |> sendType @String |> send ""
                 |> select More |> sendType @Int
                 |> send (\(x:String) (y:Int) -> x ++ showInt y) |> send 5
                 |> select More |> sendType @Bool
                 |> send (\(x:String) (y:Bool) -> x ++ showBool y) |> send True
                 |> select Done
                 |> receive
  in close c ; x
type Dual Fold ≃ ∃a . !a ; μc.+{More: ∃b . !(a -> b -> a) ; !b ; c, Done: ?a ; Close}
```

A consumer of type Fold

```
fold : Fold -> ()
fold (@a, (?x ; c)) = fold' x c
  where
    fold' x (&Done c) = c |> send x |> wait
    fold' x (&More (@b, (?f ; ?y; c))) = fold' (f x y) c
```

```
type Fold ≃ ∀a . ?a ; μc.&{More: ∀b . ?(a -> b -> a) ; ?b ; c, Done: !a ; Wait}
```

2 _ Type equivalence

Kinds

* is a non-terminal

$*$ ::=		Kind of proper types:
	T	functional
	S	session
κ ::=		Kind:
	$*$	kind of proper types
	$\kappa \Rightarrow \kappa$	kind of type operators

The syntax of types

Cf.
Alonzo Church. 1940. A Formulation of the Simple Theory of Types

T ::=		Type:
	ι	type constant
	α_{κ}	type variable
	$\lambda\alpha_{\kappa}.T$	type-level abstraction
	TT	type-level application

Type constants

$\sharp \; ::= \; ? \mid \, !$	Polarity:
$(\!\mid\!\,) \; ::= \; \{\} \mid \langle\rangle$	Record, variant
$\iota \; ::=$	Type constant:
$\rightarrow_{*\Rightarrow *'\Rightarrow \mathrm{T}}$	arrow
$(\!\mid \overline{\ell} \mid\!)_{\overline{*\Rightarrow \mathrm{T}}}$	record, variant
$\mathsf{Msg}^{\sharp}{}_{*\Rightarrow \mathrm{S}}$	message (input, output)
$\sharp\{\overline{\ell}\}_{\overline{\mathrm{S}\Rightarrow \mathrm{S}}}$	choice (external, internal)
$;_{\mathrm{S}\Rightarrow \mathrm{S}\Rightarrow \mathrm{S}}$	sequential composition
$\mathsf{End}^{\sharp}_{\mathrm{S}}$	end (wait, close)
$\mathsf{Skip}_{\mathrm{S}}$	skip
$\mathsf{Dual}_{\mathrm{S}\Rightarrow \mathrm{S}}$	dual operator
$\exists\!\forall^{\sharp}_{(\kappa\Rightarrow *)\Rightarrow *}$	quantifier (universal, existential)
$\mu_{(\kappa\Rightarrow\kappa)\Rightarrow\kappa}$	recursive type
Void_{κ}	void

Some laws

(1) $(\lambda\alpha.T)\,U \simeq T[U/\alpha]$

(2) $\mu F \simeq F(\mu F)$

(3) $\mu(\lambda\alpha.T) \simeq T$ *if* $\alpha \notin \mathrm{fv}(T)$

(4) $\mathsf{Skip};T \simeq T$

(5) $\sharp\{\overline{l\colon\ T}\};U \simeq \sharp\{\overline{l\colon\ T;U}\}$

(6) $(\exists\!\!\!\forall^{\sharp} F);U \simeq \exists\!\!\!\forall^{\sharp}\,\lambda\alpha.(F\alpha;U)$ *if* $\alpha \notin \mathrm{fv}(F,U)$

(7) $\mathsf{Dual}\,\mathsf{Skip} \simeq \mathsf{Skip}$

(8) $\mathsf{Dual}\,\mathsf{End}^{\sharp} \simeq \mathsf{End}^{\sharp^{\perp}}$

(9) $\mathsf{Dual}\,\mathsf{Void} \simeq \mathsf{Void}$

(10) $\mathsf{Dual}\,(\mathsf{Msg}^{\sharp}\,T) \simeq \mathsf{Msg}^{\sharp^{\perp}}\,T$

(11) $\mathsf{Dual}\,(\sharp\{\overline{l\colon\ T}\}) \simeq \sharp^{\perp}\{\overline{l\colon\ \mathsf{Dual}\,T}\}$

(12) $\mathsf{Dual}\,(T;U) \simeq \mathsf{Dual}\,T; \mathsf{Dual}\,U$

(13) $\mathsf{Dual}\,(\mathsf{Dual}\,T) \simeq T$

(14) $\mathsf{Dual}\,(\exists\!\!\!\forall^{\sharp} F) \simeq \exists\!\!\!\forall^{\sharp^{\perp}}\,\lambda\alpha.\mathsf{Dual}(F\alpha)$ *if* $\alpha \notin \mathrm{fv}(F)$

(15) $\mathsf{End}^{\sharp};T \simeq \mathsf{End}^{\sharp}$ *and* $\mathsf{Void};T \simeq \mathsf{Void}$

(16) $F \simeq \lambda\alpha.F\alpha$ *if* $\alpha \notin \mathrm{fv}(F)$

(17) $(T;U);V \simeq T;U;V$

Type equivalence is a weak bisimulation

Bisimulation

- Introduced by Park in the early eighties, popularised by Milner (CCS, pi-calculus) as a behavioural equivalence for processes
- Used by Amadio and Cardelli in Subtyping recursive types (1993)

Type equivalence is a weak bisimulation

- Includes internal actions:
  - Syntactically rearrange the type into a whnf, exposing a visible action
  - Aka type reduction
  - Not to be matched by the other type
- And external actions:
  - To be matched by the other type

Some invisible actions (the tau rules)

Call by name and unfold

R-β

$$(\lambda\alpha.T)\,U \xrightarrow{\tau} T[U/\alpha]$$

R-AppL

$$\frac{F \xrightarrow{\tau} G}{FT \xrightarrow{\tau} GT}$$

R-μ

$$\mu F \xrightarrow{\tau} F(\mu F)$$

Some invisible actions (the tau rules)

The semicolon rules

R-SNeut
$$\mathsf{Skip}; T \xrightarrow{\tau} T$$

R-SAssoc
$$\frac{T = \mathsf{End}^{\sharp}, \mathsf{Void}, \mathsf{Msg}^{\sharp}\, T_1, \alpha\, T_1 \cdots T_m, \mathsf{Dual}\,(\alpha\, T_1 \cdots T_m)}{(T; U); V \xrightarrow{\tau} T; (U; V)}$$

R-SChoiceDist
$$\sharp\{\overline{\ell : T}\}; U \xrightarrow{\tau} \sharp\{\overline{\ell : T; U}\}$$

R-SQuantDist
$$\frac{\alpha \notin \mathrm{fv}(F) \cup \mathrm{fv}(U)}{(\exists\!\forall^{\sharp}\, F); U \xrightarrow{\tau} \exists\!\forall^{\sharp}\, \lambda\alpha.(F\alpha; U)}$$

R-SSemiL
$$\frac{T \xrightarrow{\tau} V}{T; U \xrightarrow{\tau} V; U}$$

Some action rules

Fully applied type constructors

- Each fully applied type constructor of arity *m* shall have *m+1* transitions

$$\textsc{A-Const}\quad \frac{\iota = \rightarrow, (\!|\bar{\ell}|\!), \mathsf{Msg}^{\#}, \#\{\bar{\ell}\}, \mathsf{End}^{\#}, \exists\!\!\!/^{\#}}{(\iota\, T_1 \cdots T_m)_{*} \xrightarrow{\iota} \mathsf{Skip}}$$

$$\textsc{A-ConstArg}\quad \frac{\iota = \rightarrow, (\!|\bar{\ell}|\!), \mathsf{Msg}^{\#}, \#\{\bar{\ell}\}, \exists\!\!\!/^{\#}}{(\iota\, T_1 \cdots T_m)_{*} \xrightarrow{j} T_j}$$

$m \geq 0$ and $1 \leq j \leq m$

Some action rules

Higher-order types

- Higher-order types shall have an infinite number of transitions
- One for each variable of the appropriate kind

A-ABS

$$F_{\kappa \Rightarrow \kappa'} \xrightarrow{\lambda \alpha_\kappa} F_{\kappa \Rightarrow \kappa'}\, \alpha_\kappa$$

N.B.: Yields infinitely branching bisimulations

Some action rules

What if reduction diverges?

- Then the type transits to Skip by a Void action

$$\textsc{A-Diverge}\quad \frac{T_* \text{ diverges}}{T_* \xrightarrow{\mathsf{Void}_*} \mathsf{Skip}}$$

The diverges predicate (coinductive)

$$\textsc{D-Void}\quad \mathsf{Void}_* \text{ diverges}$$

$$\textsc{D-App}\quad \frac{F\alpha \text{ diverges}}{F \text{ diverges}}$$

$$\textsc{D-Red}\quad \frac{T_* \xrightarrow{\tau} U_* \qquad U_* \text{ diverges}}{T_* \text{ diverges}}$$

Example

Various internal reductions followed by external actions

- Take $T \triangleq \mu\,(\lambda\beta.(\&\{\mathsf{More}: (\forall\alpha.?\alpha), \mathsf{Done}: \mathsf{Wait}\}); \beta)$
- Then

$$
\begin{aligned}
T &\xrightarrow{\tau} (\lambda\beta.(\&\{\mathsf{More}: (\forall\alpha.?\alpha), \mathsf{Done}: \mathsf{Wait}\}); \beta)\,T && \text{(R-}\mu\text{)}\\
&\xrightarrow{\tau} \&\{\mathsf{More}: (\forall\alpha.?\alpha), \mathsf{Done}: \mathsf{Wait}\}; T && \text{(R-}\beta\text{)}\\
&\xrightarrow{\tau} \&\{\mathsf{More}: (\forall\alpha.?\alpha); T, \mathsf{Done}: \mathsf{Wait}; T\} && \text{(R-SChoiceDist)}
\end{aligned}
$$

$$
\swarrow_{\&\{\mathsf{More},\mathsf{Done}\}} \qquad \downarrow_1 \qquad \searrow_2
$$

$$
\mathsf{Skip} \qquad (\forall\alpha.?\alpha); T \qquad \mathsf{Wait}; T
$$

3 _ Type equivalence

Difficulty #1 _ Dealing with infinitely branching bisims

- Recall A-Abs
- One transition per variable of kind k

A-Abs

$$F_{\kappa \Rightarrow \kappa'} \xrightarrow{\lambda \alpha_\kappa} F_{\kappa \Rightarrow \kappa'}\, \alpha_\kappa$$

- **Solution**: we only need transitions by **two** (fixed) variables
- Fix two variables for each kind (vars that may appear in types)

A-Abs1

$$F_{\kappa \Rightarrow \kappa'} \xrightarrow{\lambda \alpha_{1\kappa}} F_{\kappa \Rightarrow \kappa'}\, \alpha_{1\kappa}$$

A-Abs2

$$F_{\kappa \Rightarrow \kappa'} \xrightarrow{\lambda \alpha_{2\kappa}} F_{\kappa \Rightarrow \kappa'}\, \alpha_{2\kappa}$$

- Show that $T \simeq U$ iff $T \simeq_{\mathsf{f}} U$

Difficulty #2 _ Determining when a type diverges

- Recall

The diverges predicate (coinductive)

D-VOID
$$\mathsf{Void}_{*} \text{ diverges}$$

D-APP
$$\frac{F\alpha \text{ diverges}}{F \text{ diverges}}$$

D-RED
$$\frac{T_{*} \xrightarrow{\tau} U_{*} \qquad U_{*} \text{ diverges}}{T_{*} \text{ diverges}}$$

- **Solution**: Restrict the recursion operator to base kinds $\mu_{(\mathrm{T}\Rightarrow\mathrm{T})\Rightarrow\mathrm{T}}$ and $\mu_{(\mathrm{S}\Rightarrow\mathrm{S})\Rightarrow\mathrm{S}}$
- Then divergence becomes tractable
- N.B.: Higher-order recursion is theoretically possible, its equivalence is tied to the bisimilarity of deterministic pushdown automata

Difficulty #3 _ Bisimulations may be infinite relations

Sequential composition of *n* !Bool actions

- Take $T \triangleq \mu\lambda\alpha.?\mathsf{Int}; \alpha$

  $U \triangleq \mu\lambda\beta.?\mathsf{Int}; \beta; !\mathsf{Bool}$

- Then $T \simeq U$ but the bisimulation is infinite: $\{(T, U;!\mathsf{Bool}^n), (\mathsf{Int}, \mathsf{Int}), (\mathsf{Skip}, \mathsf{Skip}) \mid n \geq 0\}$

- **Solution**:
  - Translate types to (simple) grammars
  - Run a (polynomial) algorithm to check whether two words in a simple grammar are bisimilar

FreeST 3.2 $F^{\mu;}$

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

Up and running

FreeST 5.0 $F_{\omega}^{\mu_{*};}$

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

Coming soon

Extra material

Embracing non-contractivity

- Recall `type IRepeat a = a ; IRepeat a`
- Then
$$\texttt{IRepeat}\ \mathsf{Skip} \simeq \mu\beta.\mathsf{Skip}; \beta$$
$$\simeq \mathsf{Skip}; \mu\beta.\mathsf{Skip}; \beta.$$
$$\simeq \mu\beta.\mathsf{Skip}; \beta$$
- Instead of ruling out type `IRepeat` we deem `IRepeat` Skip' well formed
- We further introduce a family of types Void_κ, one for each kind k
- Then `IRepeat` $\mathsf{Skip} \simeq \mathsf{Void}_S$:

The full set of tau rules

R-β
$$(\lambda\alpha.T)U \xrightarrow{\tau} T[U/\alpha]$$

R-μ
$$\mu F \xrightarrow{\tau} F(\mu F)$$

R-Void
$$\mathsf{Void}_{\kappa\Rightarrow\kappa'}\, T \xrightarrow{\tau} \mathsf{Void}_{\kappa'}$$

R-AppL
$$\frac{F \xrightarrow{\tau} G}{FT \xrightarrow{\tau} GT}$$

R-SNeut
$$\mathsf{Skip}; T \xrightarrow{\tau} T$$

R-SAssoc
$$\frac{T = \mathsf{End}^{\sharp}, \mathsf{Void}, \mathsf{Msg}^{\sharp}\, T_1, \alpha T_1 \cdots T_m, \mathsf{Dual}\,(\alpha T_1 \cdots T_m)}{(T;U);V \xrightarrow{\tau} T;(U;V)}$$

R-SChoiceDist
$$\sharp\{\overline{\ell : T}\};U \xrightarrow{\tau} \sharp\{\overline{\ell : T;U}\}$$

R-SQuantDist
$$\frac{\alpha \notin \mathrm{fv}(F) \cup \mathrm{fv}(U)}{(\exists\!\!\!\forall^{\sharp} F);U \xrightarrow{\tau} \exists\!\!\!\forall^{\sharp}\, \lambda\alpha.(F\alpha;U)}$$

R-SSemiL
$$\frac{T \xrightarrow{\tau} V}{T;U \xrightarrow{\tau} V;U}$$

R-DSkip
$$\mathsf{Dual}\,\mathsf{Skip} \xrightarrow{\tau} \mathsf{Skip}$$

D-Void
$$\mathsf{Dual}\,\mathsf{Void} \xrightarrow{\tau} \mathsf{Void}$$

R-DEnd
$$\mathsf{Dual}\,\mathsf{End}^{\sharp} \xrightarrow{\tau} \mathsf{End}^{\sharp^{\perp}}$$

R-DMsg
$$\mathsf{Dual}(\mathsf{Msg}^{\sharp}\, T) \xrightarrow{\tau} \mathsf{Msg}^{\sharp^{\perp}}\, T$$

R-DChoice
$$\mathsf{Dual}\,(\sharp\{\overline{\ell : T}\}) \xrightarrow{\tau} \sharp^{\perp}\{\overline{\ell : \mathsf{Dual}\, T}\}$$

R-DQuant
$$\frac{\alpha \notin \mathrm{fv}(F)}{\mathsf{Dual}(\exists\!\!\!\forall^{\sharp} F) \xrightarrow{\tau} \exists\!\!\!\forall^{\sharp^{\perp}}\, \lambda\alpha.\mathsf{Dual}(F\alpha)}$$

R-DSemi
$$\mathsf{Dual}\,(T;U) \xrightarrow{\tau} \mathsf{Dual}\,T; \mathsf{Dual}\,U$$

R-DDual
$$\mathsf{Dual}\,(\mathsf{Dual}\,T) \xrightarrow{\tau} T$$

R-DAppR
$$\frac{T \neq T_1;T_2, \mathsf{Dual}\,T_1 \qquad T \xrightarrow{\tau} U}{\mathsf{Dual}\,T \xrightarrow{\tau} \mathsf{Dual}\,U}$$

The full set of action rules

A-Const

$$\frac{\iota = \to,\ (\!|\,\bar{\ell}\,|\!),\ \mathsf{Msg}^{\#},\ \#\{\bar{\ell}\},\ \mathsf{End}^{\#},\ \exists\!\!\!/^{\#}}{(\iota\, T_1 \cdots T_m)_{*} \xrightarrow{\iota} \mathsf{Skip}}$$

A-ConstArg

$$\frac{\iota = \to,\ (\!|\,\bar{\ell}\,|\!),\ \mathsf{Msg}^{\#},\ \#\{\bar{\ell}\},\ \mathsf{End}^{\#},\ \exists\!\!\!/^{\#}}{(\iota\, T_1 \cdots T_m)_{*} \xrightarrow{j} T_j}$$

A-Var

$$(\alpha\, T_1 \cdots T_m)_{*} \xrightarrow{\alpha} \mathsf{Skip}$$

A-VarArg

$$(\alpha\, T_1 \cdots T_m)_{*} \xrightarrow{j} T_j$$

A-Dual

$$\mathsf{Dual}\,(\alpha\, T_1 \cdots T_m) \xrightarrow{\alpha^{\perp}} \mathsf{Skip}$$

A-DualArg

$$\mathsf{Dual}\,(\alpha\, T_1 \cdots T_m) \xrightarrow{j} T_j$$

A-Abs

$$F_{\kappa \Rightarrow \kappa'} \xrightarrow{\lambda \alpha_{\kappa}} F_{\kappa \Rightarrow \kappa'}\, \alpha_{\kappa}$$

A-Diverge

$$\frac{T_{*}\ \text{diverges}}{T_{*} \xrightarrow{\mathsf{Void}_{*}} \mathsf{Skip}}$$

A-SVoid

$$\mathsf{Void}; T \xrightarrow{\mathsf{Void}} \mathsf{Skip}$$

A-SEnd

$$\mathsf{End}^{\#}; T \xrightarrow{\mathsf{End}^{\#}} \mathsf{Skip}$$

A-SVar

$$(\alpha\, T_1 \cdots T_m); U \xrightarrow{\alpha} U$$

A-SVarArg

$$(\alpha\, T_1 \cdots T_m); U \xrightarrow{j} T_j$$

A-SMsg

$$\mathsf{Msg}^{\#}\, T; U \xrightarrow{\mathsf{Msg}^{\#}} U$$

A-SMsgArg

$$\mathsf{Msg}^{\#}\, T; U \xrightarrow{1} T$$

A-SDual

$$(\mathsf{Dual}\,(\alpha\, T_1 \cdots T_m)); U \xrightarrow{\alpha^{\perp}} U$$

A-SDualArg

$$(\mathsf{Dual}\,(\alpha\, T_1 \cdots T_m)); U \xrightarrow{j} T_j$$

Where $m \geq 0$ and $1 \leq j \leq m$ in all rules.

Example of divergence I

$T \triangleq \mu(\lambda\alpha_{\mathsf{s}\Rightarrow\mathsf{s}}.\alpha) \,:\, \mathsf{s}\Rightarrow\mathsf{s}$

$$
\dfrac{
\dfrac{}{T \xrightarrow{\tau} (\lambda\alpha.\alpha)\,T}\;\text{R-}\mu
\qquad
\dfrac{
\dfrac{}{(\lambda\alpha.\alpha)\,T \xrightarrow{\tau} T}\;\text{R-}\beta
\qquad
\begin{array}{c}\vdots\\ T\ \text{diverges}\end{array}
}{(\lambda\alpha.\alpha)\,T\ \text{diverges}}\;\text{D-Red}
}{T\ \text{diverges}}\;\text{D-Red}
$$

Example of divergence II

$\mathsf{Void}_{s \Rightarrow s}$

$$
\dfrac{
  \dfrac{}{\mathsf{Void}_{s \Rightarrow s}\ \delta \xrightarrow{\tau} \mathsf{Void}_{s}}\ \textsc{R-Void}
  \qquad
  \dfrac{}{\mathsf{Void}_{s}\ \text{diverges}}\ \textsc{D-Void}
}{
  \dfrac{\mathsf{Void}_{s \Rightarrow s}\ \delta\ \text{diverges}}{\mathsf{Void}_{s \Rightarrow s}\ \text{diverges}}\ \textsc{D-App}
}\ \textsc{D-Red}
$$

Example of divergence III

$\lambda\gamma_{\mathsf{s}}.\mu F : \mathsf{s} \Rightarrow \mathsf{s}$ where $F \triangleq \lambda\beta_{\mathsf{s}}.\beta;\gamma$.

$$
\dfrac{
  \dfrac{
    \dfrac{}{(\lambda\gamma.\mu F)\gamma \xrightarrow{\tau} \mu F}\,\text{R-}\beta
    \qquad
    \dfrac{
      \dfrac{}{\mu F \xrightarrow{\tau} F(\mu F)}\,\text{R-}\mu
      \qquad
      \dfrac{
        \dfrac{\dfrac{}{\mu F \xrightarrow{\tau} F(\mu F)}\,\text{R-}\mu}{(\mu F);\gamma \xrightarrow{\tau} (F(\mu F));\gamma}\,\text{R-SSemiL}
        \qquad
        \dfrac{\vdots \qquad \begin{array}{c}\vdots\\ ((\mu F);\gamma);\gamma\end{array}}{(F(\mu F));\gamma \text{ diverges}}\,\text{D-Red}
      }{(\mu F);\gamma \text{ diverges}}\,\text{D-Red}
    }{\mu F \text{ diverges}}\,\text{D-Red}
  }{(\lambda\gamma.\mu F)\gamma \text{ diverges}}\,\text{D-Red}
}{\lambda\gamma.\mu F \text{ diverges}}\,\text{D-App}
$$

Weak bisimulation

Definition 4.3 (Weak bisimulation and weak bisimilarity). A weak bisimulation is a relation $R \subseteq F^{\mu;}_{\omega} \times F^{\mu;}_{\omega}$ such that, for every $(T, U) \in R$ and every transition label a,

- **(zig)** if $T \xrightarrow{a} T'$, then there exists U' such that $U \overset{a}{\Longrightarrow} U'$ and $(T', U') \in R$;
- **(zag)** if $U \xrightarrow{a} U'$, then there exists T' such that $T \overset{a}{\Longrightarrow} T'$ and $(T', U') \in R$;

Types T, U are said to be weakly bisimilar, denoted $T \simeq U$, if there exists a weak bisimulation R such that $(T, U) \in R$.

Translation to grammar

$$
\begin{array}{rcll}
\mathsf{word}(\mathsf{Skip}) & = & \varepsilon & \text{(W-Skip)} \\
\mathsf{word}(\iota) \mid \iota = \mathsf{End}^{\sharp}, \mathsf{Void}_{*} & = & X \text{ with } X \xrightarrow{\iota} \bot & \text{(W-EndVoid)} \\
\mathsf{word}(\mathsf{Msg}^{\sharp}\, T) & = & X \text{ with } X \xrightarrow{\mathsf{Msg}^{\sharp}} \varepsilon \text{ and } X \xrightarrow{1} \mathsf{word}(T)\bot & \text{(W-Msg)} \\
\mathsf{word}(T;U) & = & \mathsf{word}(T)\ \mathsf{word}(U) & \text{(W-Seq)} \\
\mathsf{word}(\mathsf{Dual}\,(\alpha\, T_1 \cdots T_m)) & = & X \text{ with } X \xrightarrow{\alpha^{\bot}} \varepsilon \text{ and } X \xrightarrow{j} \mathsf{word}(T_j)\bot & \text{(W-DualVar)} \\
\mathsf{word}((\iota T_1 \cdots T_m)_{*}) \mid \iota = \rightarrow, (\!|\, \overline{\ell}\,|\!), \#\{\overline{\ell}\}, \exists\!\!\not\,^{\sharp} & = & X \text{ with } X \xrightarrow{\iota} \bot \text{ and } X \xrightarrow{j} \mathsf{word}(T_j) & \text{(W-Const)} \\
\mathsf{word}((\alpha T_1 \cdots T_m)_{*}) & = & X \text{ with } X \xrightarrow{\alpha} \varepsilon \text{ and } X \xrightarrow{j} \mathsf{word}(T_j)\bot & \text{(W-Var)} \\
\mathsf{word}(T) \mid \mu\mathsf{redex}(T) \text{ defined} \wedge T \Downarrow \mathsf{Skip} & = & \varepsilon & \text{(W-}\mu\text{Skip)} \\
\mathsf{word}(T) \mid \mu\mathsf{redex}(T) \text{ defined} \wedge T \Downarrow U & = & X \text{ with } X \xrightarrow{a} \gamma\delta, & \text{(W-}\mu\text{NSkip)} \\
 & & \text{where } Z\delta = \mathsf{word}(U) \text{ and } Z \xrightarrow{a} \gamma & \\
\mathsf{word}(F_{\kappa \Rightarrow \kappa'}) & = & X \text{ with } X \xrightarrow{\lambda\alpha_{\kappa}} \mathsf{word}(F\alpha_{\kappa}) \text{ and } X \xrightarrow{\lambda\beta_{\kappa}} \mathsf{word}(F\beta_{\kappa}) & \\
 & & & \text{(W-Abs)} \\
\mathsf{word}(T) \mid T \xrightarrow{\tau} U & = & \mathsf{word}(U) & \text{(W-}\tau\text{)}
\end{array}
$$