

# 1 Languages

## 1.1 Flexible Language

Flexible Language:

- Non deterministic semantic, with branches that reduce to a value and other ending in error.
- Type error means stuck.
- Without any type information in the syntax of the language.
- Uses the explicit substitution in `mlet`, and the implicit substitution in `lambdas`. In the case of the `mlet` is used the explicit substitution because the implicit substitution of a variable by a value would eliminate the overloading.

Characterization of the errors for Flexible Language:

- Free variable error is detected if a variable is evaluated in the empty environment.
- Type error is detected if the operators `not` or `add1` are applied with parameters that are not boolean or numeric value, respectively. Also, if the left side of the function application is not a lambda.

$t ::=$	terms	$op ::=$	operators
$b$	boolean value	<code>add1</code>	sum
$n$	numeric value	<code>not</code>	negation
$op$	operator	$c ::=$	configurations
$x$	variable	$v$	
$\lambda x. t$	abstraction	$t[s]$	
$t t$	application	$c c$	
<code>mlet <math>x = t</math> in <math>t</math></code>	overloading let	<code>mlet <math>x = c</math> in <math>c</math></code>	
$v ::=$	values	$e$	
$b$	boolean value		
$n$	numeric value	$e ::=$	errors
$op$	operator	<code>NameError</code>	name error
$(\lambda x. t)[s]$	closure	<code>TagError</code>	type error
$b ::=$	boolean value	$s ::=$	explicit substitutions
<code>true</code>	true value	$\bullet$	empty substitution
<code>false</code>	false value	$x \mapsto \{\bar{v}\}, s$	variable substitution

Figure 1: Syntax of the Flexible Language.

	$c \longrightarrow c$
$b[s] \longrightarrow b$	(False)
$n[s] \longrightarrow n$	(Num)
$op[s] \longrightarrow op$	(Op)
$x[x \mapsto \{\bar{v}\}, s] \longrightarrow v_i$	(VarOk)
$\frac{x \neq y}{x[y \mapsto \{\bar{v}\}, s] \longrightarrow x[s]}$	(VarNext)
$\text{mlet } x = v \text{ in } t_2[s] \longrightarrow t_2[x \mapsto v \oplus s]$	(Let)
$(\lambda x. t_2)[s] v \longrightarrow ([x \mapsto v]t_2)[s]$	(App)
$\text{add1 } n \longrightarrow n + 1$	(Sum)
$\text{not } b \longrightarrow \neg b$	(Negation)

Figure 2: Reduction rules for Flexible Language.

$(\text{mlet } x = t_1 \text{ in } t_2)[s] \longrightarrow \text{mlet } x = t_1[s] \text{ in } t_2[s]$	(LetSub)
$(t_1 t_2)[s] \longrightarrow t_1[s] t_2[s]$	(AppSub)

Figure 3: Substitution rules for Flexible Language.

**Definition 1** ( $\oplus$ ). *Given an environment  $s$  and a variable binding  $x \mapsto v_1$ , the operator  $\oplus$  is defined as follows:*

$$s \oplus x \mapsto v_1 = \begin{cases} x \mapsto \{v_1\} & s = \emptyset \\ x \mapsto \{\bar{v}\} \cup \{v_1\}, s' & s = x \mapsto \{\bar{v}\}, s' \\ y \mapsto \{\bar{v}\}, s' \oplus x \mapsto v_1 & s = y \mapsto \{\bar{v}\}, s' \end{cases}$$

## 1.2 Tag Driven Language

Tag Driven Language:

- Non deterministic semantic, with branches that reduce to a value and other ending in error.
- Type error means stuck.
- Dispatch error means stuck.
- Without any type information in the syntax of the language.
- Semantic "tag driven", introducing flat tag.

Characterization of the errors for Tag Driven Language:

$$\begin{array}{c}
\frac{c_1 \longrightarrow c'_1}{\text{mlet } x = c_1 \text{ in } c_2 \longrightarrow \text{mlet } x = c'_1 \text{ in } c_2} \quad (\text{Let1}) \\
\\
\frac{c_1 \longrightarrow c'_1}{c_1 \ c_2 \longrightarrow c'_1 \ c_2} \quad (\text{App1}) \\
\\
\frac{c \longrightarrow c'}{v \ c \longrightarrow v \ c'} \quad (\text{App2})
\end{array}$$

Figure 4: Congruence rules for Flexible Language.

$$\begin{array}{c}
\text{mlet } x = e \text{ in } c_2 \longrightarrow e \quad (\text{LetErr}) \\
\\
e \ c_2 \longrightarrow e \quad (\text{AppErr1}) \\
\\
v \ e \longrightarrow e \quad (\text{AppErr2})
\end{array}$$

Figure 5: Propagation error rules for Flexible Language.

- Free variable and type error are detected in the same cases that the Flexible Language.
- Dispatch error is detected if the operators `not` or `add1` are applied with overloaded parameters that do not have instance with tag `Bool` or `Int` in the environment, respectively. Also, if the left side of the function application is an overloaded variable, but does not exist an instance with tag `Fun` in the environment.

**Definition 2** (lookup). *The relation lookup is defined as follows:*

$$\text{lookup} = \{(x, s, S, v) \mid x \mapsto v \in \text{flat}(s) \wedge \text{tag}(v) = S\}$$

**Definition 3** (flat). *The function flat is defined as follows:*

$$\text{flat}(s) = \begin{cases} \emptyset & s = \emptyset \\ x \mapsto v_1 \cdots, x \mapsto v_n, \text{flat}(s') & s = x \mapsto \{\bar{v}\}, s' \end{cases}$$

**Definition 4** (tag). *The function tag is defined as follows:*

$$\text{tag}(v) = \begin{cases} \text{Int} & v = n \\ \text{Bool} & v = b \\ \text{Fun} & v = \lambda x. t \end{cases}$$

### 1.3 Tag Driven Language with ascription

### 1.4 Strict Language

Strict Language:

- Deterministic semantic. With the use of multi-values a program can reduce to a set of value.

$x[] \longrightarrow \text{NameError}$	(NameError)
$\frac{v_1 \neq \lambda x. t}{v_1 v_2 \longrightarrow \text{TagError}}$	(TypeErrApp)
$\frac{v \neq n}{\text{add1 } v \longrightarrow \text{TagError}}$	(TypeErrSum)
$\frac{v \neq b}{\text{not } v \longrightarrow}$	(TypeErrNegation)

Figure 6: Error rules for Flexible Language.

$S$	::=	tags
	Int	integer tag
	Bool	boolean tag
	Fun	function tag
$e$	::=	errors
	NameError	name error
	TagError	type error
	DispatchError	dispatch error
...		

Figure 7: Syntax of the Tag Driven Language.

- Type error means stuck.
- Dispatch error means stuck.
- Ambiguity error means stuck.

Characterization of the errors for Strict Language:

- Free variable and type error are detected in the same cases that the Tag Driven Language with ascription.
- Ambiguity error is detected if the left side of the function application have more than one instance with **Fun** tag or in the right side have more than one value for the application. Strict detection of ambiguity.

The only rule that change is:  $\frac{c \longrightarrow c'}{w \ c \longrightarrow w \ c'} \text{ (App2)}$

**Definition 5** ( $\oplus$ ). *Given an environment  $s$  and a variable binding  $x \mapsto v_1$ , the operator  $\oplus$  is defined as follows:*

$$s \oplus x \mapsto w = \begin{cases} x \mapsto w & s = \emptyset \\ x \mapsto w' \cup w, s' & s = x \mapsto w', s' \\ y \mapsto w', s' \oplus x \mapsto w & s = y \mapsto w', s' \end{cases}$$

**Definition 6** ( $\text{filter}(\cdot, \cdot)$ ). *The function filter is defined as follows:*

$$\text{filter}(w, S) = \{v \mid v \in w \wedge \text{tag}(v) = S\}$$

$\frac{\dots \text{lookup}(x_1, [s_1], \text{Fun}, v_1)}{x_1[s_1] \ v_2 \longrightarrow v_1 \ v_2}$	(AppVar)
$\frac{\text{lookup}(x, [s], \text{Int}, n)}{\text{add1 } x[s] \longrightarrow \text{add1 } n}$	(SumVar)
$\frac{\text{lookup}(x, [s], \text{Bool}, b)}{\text{not } x[s] \longrightarrow \text{not } b}$	(NegationVar)

Figure 8: Reduction rules for Tag Driven Language.

$\frac{c_1 \longrightarrow c'_1 \quad \text{notVal}(c_1)}{\text{mlet } x = c_1 \text{ in } c_2 \longrightarrow \text{mlet } x = c'_1 \text{ in } c_2}$	(Let1)
$\frac{c_1 \longrightarrow c'_1 \quad \text{notVal\_Var}(c_1)}{c_1 \ c_2 \longrightarrow c'_1 \ c_2}$	(App1)
$\frac{c \longrightarrow c' \quad \text{notVal}(c)}{(\lambda x. t_2)[s] \ c \longrightarrow (\lambda x. t_2)[s] \ c'}$	(App2)
$\frac{c_2 \longrightarrow c'_2 \quad \text{notVal}(c_2)}{x[s] \ c_2 \longrightarrow x[s] \ c'_2}$	(App3)
$\frac{c \longrightarrow c' \quad \text{notVal\_Var}(c)}{op \ c \longrightarrow op \ c'}$	(App4)

Figure 9: Congruence rules for Tag Driven Language.

## 1.5 Overloading Language

- Deterministic semantic. With the use of multi-values a program can reduce to a set of value.
- Type error detection.
- Dispatch error detection.
- Ambiguity error detection.
- Type annotation in lambda functions, `mlet` and ascription.
- More expressive than Strict Language, with the use structural tags.
- Do not support context-dependent overloading.
- Como no esta verificacda la informacion de tipo, no se puede decir nada acerca de la semantica.

**Definition 7 (tag).** *The function tag is defined as follows:*

$$\text{tag}(v) = \begin{cases} \text{Int} & v = n \\ \text{Bool} & v = b \\ T_1 \rightarrow T_2 & v = ((\lambda x. t_2)^{T_1 \rightarrow T_2})[s] \end{cases}$$

$\frac{\dots}{\neg\text{lookup}(x_1, [s_1], \text{Fun}, v_1)}$	
$\frac{x_1[s_1] \ v_2 \longrightarrow \text{DispatchError}}{\neg\text{lookup}(x_1, [s_1], \text{Fun}, v_1)}$	(DisErrApp)
$\frac{\neg\text{lookup}(x, [s], \text{Int}, n)}{\text{add1 } x[s] \longrightarrow \text{DispatchError}}$	(DisErrSum)
$\frac{\neg\text{lookup}(x, [s], \text{Bool}, b)}{\text{not } x[s] \longrightarrow \text{DispatchError}}$	(DisErrNegation)

Figure 10: Error rules for Tag Driven Language.

$\dots$		
$t$	$::=$	terms
	$\dots$	
	$t :: S$	ascription
$c$	$::=$	configurations
	$\dots$	
	$c :: S$	

Figure 11: Syntax of the Tag Driven Language with ascriptions.

**Definition 8** ( $\text{filter}(\cdot, \cdot)$ ). *The function filter is defined as follows:*

$$\text{filter}(w, T) = \{v \mid v \in w \mid \text{tag}(v) = T\}$$

**Definition 9** ( $\text{lookup}$ ). *The function lookup is defined as follows:*

$$\text{lookup}(w_1, w_2) = \{(v_1, v_2) \mid v_1 \in w_1 \wedge v_2 \in w_2 \wedge \text{tag}(v_1) = T_1 \rightarrow T_2 \wedge \text{tag}(v_2) = T_1\}$$

## 1.6 Static semantic for Overloading Language

**Definition 10** ( $\oplus$ ). *Given a multi-type context  $\phi$  and a pair  $(x : T)$ , the operator  $\oplus$  is defined as follows:*

$$\phi \oplus (x : T) = \begin{cases} x : \{T\} & \phi = \emptyset \\ \phi', x : (T^* \cup \{T\}) & \phi = \phi', x : T^* \\ \phi' \oplus (x : T), y : T^* & \phi = \phi', y : T^* \end{cases}$$

**Definition 11** ( $\Gamma(s)$ ). *The typing context built from a substitution  $s$ , writing  $\Gamma(s)$ , it is defined as follows:*

$$\Gamma(s) = \begin{cases} \emptyset & s = \bullet \\ \Gamma(s'), x : T & s = (x, v) : s' \wedge \Gamma \vdash_c v : T \end{cases}$$

	$c \longrightarrow c$
$\frac{\dots \quad \text{tag}(v) = S}{v :: S \longrightarrow v}$	(Asc)
$(t :: S)[s] \longrightarrow t[s] :: S$	(AscSub)
$\frac{\text{lookup}(x, [s], S, v)}{x[s] :: S \longrightarrow v}$	(AscVar)
$\frac{c \longrightarrow c' \quad \text{notVal\_Var}(c)}{c :: S \longrightarrow c' :: S}$	(Asc1)
$error :: S \longrightarrow error$	(AscErr)
$\frac{\text{tag}(v) \neq S}{v :: S \longrightarrow \text{TagError}}$	(TypeErrAsc)
$\frac{\neg \text{lookup}(x, [s], S, v)}{x[s] :: S \longrightarrow v}$	(DisErrAsc)

Figure 12: Rules for Tag Driven Language with ascriptions.

$w$	$::=$	$\dots$	multi – value
		$\{\bar{v}\}$	set of values
$c$	$::=$	$w$	configurations
		$t[s]$	
		$c \ c$	
		<b>mlet</b> $x = c$ <b>in</b> $c$	
		$c :: S$	
		$e$	error
$e$	$::=$	$\dots$	errors
		<b>AmbiguityError</b>	ambiguity error

Figure 13: Syntax of the Strict Language.

$c \longrightarrow c$	
$w[s] \longrightarrow w$	(MultiValue)
$x[x \mapsto w, s] \longrightarrow w$	(VarOk)
$\frac{x \neq y}{x[y \mapsto w, s] \longrightarrow x[s]}$	(VarNext)
$\frac{\text{filter}(w, S) = w' \quad w' \neq \emptyset}{w :: S \longrightarrow w'}$	(Asc)
$\text{mlet } x = w \text{ in } t_2[s] \longrightarrow t_2[x \mapsto w \oplus s]$	(Let)
$\frac{\text{filter}(w_1, \text{Fun}) = \{\lambda x. t_2\}}{w_1 \ w_2 \longrightarrow ([x \mapsto w_2]t_2)[s]}$	(App)
$\frac{\text{filter}(w, \text{Int}) = \{\bar{n}\}}{\text{add1 } w \longrightarrow \{\bar{n} + 1\}}$	(Sum)
$\frac{\text{filter}(w, \text{Bool}) = \{\bar{b}\}}{\text{not } w \longrightarrow \{\neg \bar{b}\}}$	(Negation)

Figure 14: Reduction rules for Strict Language.

$\frac{\text{filter}(w, S) = \{v\}}{w :: S \longrightarrow \{v\}}$	(Asc)
$\text{mlet } x = w \text{ in } t_2[s] \longrightarrow t_2[x \mapsto w \oplus s]$	(Let)
$\frac{\text{filter}(w_1, \text{Fun}) = \{\lambda x. t_2\}}{w_1 \ w_2 \longrightarrow ([x \mapsto w_2]t_2)[s]}$	(App)
$\frac{\text{filter}(w, \text{Int}) = \{n\}}{\text{add1 } w \longrightarrow \{n + 1\}}$	(Sum)
$\frac{\text{filter}(w, \text{Bool}) = \{b\}}{\text{not } w \longrightarrow \{\neg b\}}$	(Negation)

Figure 15: Reduction rules for Strict Language.



$\dots$		
$\frac{\text{filter}(w, S) = \{ \}}{w :: S \longrightarrow \text{DispatchError}}$		(DisErrAsc)
$\text{mlet } x = w \text{ in } t_2[s] \longrightarrow t_2[x \mapsto w \oplus s]$		(DisErrLet)
$\frac{\text{filter}(w_1, \text{Fun}) = \{ \}}{w_1 \ w_2 \longrightarrow \text{DispatchError}}$		(DisErrApp)
$\frac{\text{filter}(w, \text{Int}) = \{ \}}{\text{add1 } w \longrightarrow \text{DispatchError}}$		(DisErrSum)
$\frac{\text{filter}(w, \text{Bool}) = \{ \}}{\text{not } w \longrightarrow \text{DispatchError}}$		(DisErrNegation)
$\frac{\text{filter}(w, S) = \{ \}}{w :: S \longrightarrow \text{AmbiguityError}}$		(AmbErrAsc)
$\text{mlet } x = w \text{ in } t_2[s] \longrightarrow t_2[x \mapsto w \oplus s]$		(AmbErrLet)
$\frac{\text{filter}(w_1, \text{Fun}) = \{ \}}{w_1 \ w_2 \longrightarrow \text{AmbiguityError}}$		(AmbErrApp)
$\frac{\text{filter}(w, \text{Int}) = \{ \}}{\text{add1 } w \longrightarrow \text{AmbiguityError}}$		(AmbErrSum)
$\frac{\text{filter}(w, \text{Bool}) = \{ \}}{\text{not } w \longrightarrow \text{AmbiguityError}}$		(AmbErrNegation)

Figure 16: Error rules for Strict Language.

$t$	$::=$	terms	$v$	$::=$	values
	$b$	boolean value		$b$	boolean value
	$n$	numeric value		$n$	numeric value
	$op$	operator		$op$	operator
	$(\lambda x. t)^{T \rightarrow T}$	abstraction		$((\lambda x. t)^{T \rightarrow T})[s]$	closure
	$x$	variable			
	$t \ t$	application			
	$\text{mlet } x = t \text{ in } t$	overloading let	$c$	$::=$	configurations
	$t :: T$	ascription		$w$	
				$t[s]$	
$T$	$::=$	types		$c \ c$	
	$\text{Int}$	type of integers		$\text{mlet } x : T = c \text{ in } c$	
	$\text{Bool}$	type of booleans		$c :: T$	
	$T \rightarrow T$	type of functions			

Figure 17: Syntax of the Overloading Language.

	$c \longrightarrow c$	
$w[s] \longrightarrow w$	(MultiValue)	
$x[x \mapsto w, s] \longrightarrow w$	(VarOk)	
$\frac{x \neq y}{x[y \mapsto w, s] \longrightarrow x[s]}$	(VarNext)	
$\frac{s}{(t :: T)[s] \longrightarrow t[s] :: T}$	(AscSub)	
$(\text{mlet } x : T_1 = t_1 \text{ in } t_2)[s] \longrightarrow \text{mlet } x : T_1 = t_1[s] \text{ in } t_2[s]$	(LetSub)	
$(t_1 \ t_2)[s] \longrightarrow t_1[s] \ t_2[s]$	(AppSub)	
$\frac{\text{filter}(w, S) = \{v\}}{w :: T \longrightarrow \{v\}}$	(Asc)	
$\frac{\text{filter}(w, T_1) = \{v\}}{\text{mlet } x : T_1 = w \text{ in } t_2[s] \longrightarrow t_2[x \mapsto v \oplus s]}$	(Let)	
$\frac{(((\lambda x. t_2)^{T_1 \rightarrow T_2})[s], v_2) = \text{lookup}(w_1, w_2)}{w_1 \ w_2 \longrightarrow ([x \mapsto v_2]t_2)[s]}$	(App)	
$\frac{\text{filter}(w, \text{Int}) = \{n\}}{\text{add1 } w \longrightarrow \{n + 1\}}$	(Sum)	
$\frac{\text{filter}(w, \text{Bool}) = \{b\}}{\text{not } w \longrightarrow \{\neg b\}}$	(Negation)	
$\frac{c \longrightarrow c'}{c :: T \longrightarrow c' :: T}$	(Asc1)	
$\frac{c_1 \longrightarrow c'_1}{\text{mlet } x = c_1 \text{ in } c_2 \longrightarrow \text{mlet } x = c'_1 \text{ in } c_2}$	(Let1)	
$\frac{c_1 \longrightarrow c'_1}{c_1 \ c_2 \longrightarrow c'_1 \ c_2}$	(App1)	
$\frac{c \longrightarrow c'}{v \ c \longrightarrow v \ c'}$	(App2)	

Figure 18: Reduction rules for Overloading Language.

$T^*$	$::=$	$\dots$ $\{\bar{T}\}$	multi – types multi – type
$\Gamma$	$::=$	$\emptyset$ $\Gamma, x : T$	typing contexts empty context term variable binding
$\phi$	$::=$	$\emptyset$ $\phi, x : T^*$	multi – typing contexts empty context term variable binding

Figure 19: Syntax for Overloading Language with static semantic.

	$\Gamma; \phi \vdash t : T$
$\Gamma; \phi \vdash b : \{\text{Bool}\}$	(TBool)
$\Gamma; \phi \vdash n : \{\text{Int}\}$	(TInt)
$\Gamma; \phi \vdash \text{not} : \{\text{Bool} \rightarrow \text{Bool}\}$	(TNegation)
$\Gamma; \phi \vdash \text{add1} : \{\text{Int} \rightarrow \text{Int}\}$	(TSum)
$\frac{x : T \in \Gamma}{\Gamma; \phi \vdash x : \{T\}}$	(TVar $\Gamma$ )
$\frac{x : T^* \in \phi}{\Gamma; \phi \vdash x : T^*}$	(TVar $\phi$ )
$\frac{x \notin \text{dom}(\Gamma \cup \phi) \quad \Gamma, x : T_1; \phi \vdash t_2 : T_2^* \quad T_2 \in T_2^*}{\Gamma \vdash_c (\lambda x. t_2)^{T_1 \rightarrow T_2} : \{T_1 \rightarrow T_2\}}$	(TAbs)
$\frac{\Gamma; \phi \vdash t : T^* \quad T \in T^*}{\Gamma; \phi \vdash t :: T : \{T\}}$	(TAsc)
$\frac{x \notin \text{dom}(\Gamma) \quad \Gamma; \phi \vdash t_1 : T_1^* \quad T_1 \in T_1^* \quad \Gamma; \phi \oplus (x : T_1) \vdash t_2 : T_2^*}{\Gamma; \phi \vdash \text{mlet } x : T_1 = t_1 \text{ in } t_2 : T_2^*}$	(TLet)
$\frac{\Gamma; \phi \vdash t_1 : T_1^* \quad \Gamma; \phi \vdash t_2 : T_2^* \quad !\exists(T_1 \rightarrow T_2) \in T_1^* \mid !\exists(T_1) \in T_2^*}{\Gamma; \phi \vdash t_1 t_2 : \{T_2\}}$	(TApp)

Figure 20: Term typing rules.

$\Gamma; \phi \vdash t : T$	
$\Gamma \vdash_c b : \{\text{Bool}\}$	(CTBool)
$\Gamma \vdash_c b[s] : \{\text{Bool}\}$	(CSTBool)
$\Gamma \vdash_c n : \{\text{Int}\}$	(CTInt)
$\Gamma \vdash_c n[s] : \{\text{Int}\}$	(CSTInt)
$\Gamma \vdash_c \text{not} : \{\text{Bool} \rightarrow \text{Bool}\}$	(CTNegation)
$\Gamma \vdash_c \text{not } [s] : \{\text{Bool} \rightarrow \text{Bool}\}$	(CSTNegation)
$\Gamma \vdash_c \text{add1} : \{\text{Int} \rightarrow \text{Int}\}$	(CTSum)
$\Gamma \vdash_c \text{add1 } [s] : \{\text{Int} \rightarrow \text{Int}\}$	(CSTSum)
$\frac{x : T \in \Gamma}{\Gamma \vdash_c x[s] : \{T\}}$	(CTVar $\Gamma$ )
$\frac{x : T^* \in \phi(s)}{\Gamma \vdash_c x[s] : T^*}$	(CTVar $\phi$ )
$\frac{x \notin \text{dom}(\Gamma \cup \phi(s)) \quad \Gamma, x : T_1 \vdash_c t_2[s] : T_2^* \quad T_2 \in T_2^*}{\Gamma \vdash_c ((\lambda x. t_2)^{T_1 \rightarrow T_2})[s] : \{T_1 \rightarrow T_2\}}$	(CTAbs)
$\frac{\Gamma \vdash_c t[s] :: T : T^*}{\Gamma \vdash_c (t :: T)[s] : T^*}$	(CTAsc)
$\frac{\Gamma \vdash_c c : T^* \quad T \in T^*}{\Gamma \vdash_c c :: T : \{T\}}$	(CSTAsc)
$\frac{\Gamma \vdash_c \text{mlet } x : T_1 = t_1[s] \text{ in } t_2[s] : T^*}{\Gamma \vdash_c (\text{mlet } x : T_1 = t_1 \text{ in } t_2)[s] : T^*}$	(CTLet)
$\frac{x \notin \text{dom}(\Gamma) \quad \Gamma \vdash_c c_1 : T_1^* \quad T_1 \in T_1^* \quad \Gamma; \phi(s) \oplus (x : T_1) \vdash t_2 : T_2^*}{\Gamma \vdash_c \text{mlet } x : T_1 = c_1 \text{ in } t_2[s] : T_2^*}$	(CSTLet)
$\frac{\Gamma \vdash_c t_1[s] \ t_2[s] : T^*}{\Gamma \vdash_c (t_1 \ t_2)[s] : T^*}$	(CTCApp)
$\frac{\Gamma \vdash_c c_1 : T_1^* \quad \Gamma \vdash_c c_2 : T_2^* \quad !\exists(T_1 \rightarrow T_2) \in T_1^* \mid !\exists(T_1) \in T_2^*}{\Gamma \vdash_c c_1 \ c_2 : \{T_2\}}$	(CSTApp)

Figure 21: Configuration typing rules.