Annex

A Formal computational supertype/subtype rules

This annex forms an integral part of this Recommendation | International Standard. It defines formal subtyping rules for computational interface signatures. The types of computational interface signatures can be higher order as implied by 7.2.2.4 on parameter rules. This annex only formalizes a first-order subset of the subtyping rules — formalizing higher order features of computational interface signature types is left for further study.

The annex defines a first order type system that consists of a simple type language together with type equality rules and signature subtyping rules. It also describes a sound and complete type checking algorithm for the type system. Signal signature interface types, operation interface signature types and stre am interface signature types are defined using the type language. Since stream interface signature subtyping is only partially defined in 7.2.4.2, this annex only formalizes the subtyping rule that applies between corresponding flows.

A.1 Notations and conventions

The following notations are used

- α, β, γ, etc., denote types;
- *t*, *s*, etc., denote identifiers for types (i.e. type variables) and ground types (i.e., type constants); the set of type variables (and type constants) is called T_{var};
- *a, b, c,* a_1, a_2, a_n etc., denote identifiers or *labels* for elements of structures in the type language; the set of labels is called Λ;
- $\alpha[\beta/t]$ denotes the substitution of β for *t* in α;
- *Nil* denotes a predefined type constant.

A.2 Type system

The type system contains type constants, functions, cartesian products, records, tagged unions recursive definitions. The type language, *Type*, is given by the grammar in figure 1.

$$
\begin{array}{lll}
a & ::= & t \\
 & | & \bot \\
 & | & \top \\
 & | & \alpha \rightarrow \beta \\
 & | & \alpha_1 \times \ldots \times \alpha_n \\
 & | & \langle a_1 : \alpha_1, \ldots, a_n : \alpha_n \rangle \\
 & | & [c_1 : \gamma_1, \ldots, c_n : \gamma_n] \\
 & | & \mu t.\alpha
\end{array}
$$

Figure 1 - Abstract syntax for type declararations

The ground types are called $\top$ (top) and $\perp$ (bottom). They play the roles of greatest and least elements in the subtype relation, respectively. Functions are denoted thus: $\alpha \rightarrow \beta$. Cartesian products are denoted thus: $\alpha_1 \times ... \times \alpha_n$. Unions are denoted thus: $[c_1 : \gamma_1 ..., c_n : \gamma_n]$. Records are denoted thus: $\langle a_1 : \alpha_1, ..., a_n : \alpha_n\rangle$.

μ is a variable binding operator. Recursive types can be constructed by binding types to identifiers and referencing an identifier for one type in another.

Parentheses are used to determine precedence where necessary. In their absence, $\rightarrow$ associates to the right, and the scoping of μ extends to the right as far as possible.

The set of free variables occuring in α is denoted thus: $FV(\alpha)$.

A.2.1 Typing rules

This clause gives type equality rules and subtyping rules for the language given.

A type α is *contractive* in the type variable t, denoted $\alpha \downarrow t$, if either t does not occur free in α, or α can be rewritten via unfolding as a type of one of the following forms

— $\alpha_1 \rightarrow \alpha_2$;

— $\langle a_1 : \alpha_1, ..., a_m : \alpha_m\rangle$;

— $[c_1 : \gamma_1, ..., c_n : \gamma_n]$;

— $\alpha_1 \times ... \times \alpha_n$

The type equality rules are given in figure 2. Type equality is denoted by =.

(E.1)	$\alpha = \alpha$
(E.2)	$\alpha = \beta \Rightarrow \beta = \alpha$
(E.3)	$\alpha = \beta, \beta = \gamma \Rightarrow \alpha \Rightarrow \gamma$
(E.4)	$\alpha_1 = \alpha_2, \beta_1 = \beta_2 \Rightarrow \alpha_1 \rightarrow \beta_1 = \alpha_2 \rightarrow \beta_1$
(E.5)	$\alpha = \beta \Rightarrow \mu t.\alpha = \mu t.\beta$
(E.6)	$\forall i \in \{1, ..., n\}, \alpha_i = \beta_i \Rightarrow \alpha_1 \times ... \times \alpha_n = \beta_1 \times ... \times \beta_n$
(E.7)	$\forall i \in \{1, ..., n\}, \alpha_i = \beta_i \Rightarrow \langle a_1 : \alpha_1, ..., a_n : \alpha_n\rangle = \langle a_1 : \beta_1, ... ,a_n : \beta_n\rangle$
(E.8)	$\forall i \in \{1, ..., n\}, \alpha_i = \beta_i \Rightarrow [a_1 : \alpha_1, ..., a_n : \alpha_n] = [a_1 : \beta_1, ... ,a_n : \beta_n]$
(E.9)	$\mu t.t = \perp$
(E.10)	$\alpha[\mu t.\alpha/t] = \mu t.\alpha$
(E.11)	$\alpha[\beta/t] = \beta_1, \alpha[B_2/t] = \beta_2\ \alpha \downarrow t \Rightarrow \beta_1 = \beta_2$

Figure 2 - Type equality rules

The subtyping rules are given in the form of inference rules on judgments that resemble a Prolog program. Judgements are of the form: $\Gamma \vdash \alpha \leq \beta$, where Γ is a set of subtyping assumptions on type variables of the form: $\{t_1 \leq s_1, ..., t_n \leq s_n\}$. A typical rule may take the following form

$$\Gamma \vdash \alpha_1 \leq \beta_1, \Gamma \vdash \alpha_2 \leq \beta_2 \Rightarrow \Gamma \vdash \alpha \leq \beta.$$

Informally, this means that in order to determine whether $\Gamma \vdash \alpha \leq \beta$ holds, one must first try to determine whether $\Gamma \vdash \alpha_1 \leq \beta_1$ and $\Gamma \vdash \alpha_2 \leq \beta_2$. If these sub-goals are reached then one may conclude that α is a subtype of β.

The subtyping rules are given in figure 3. It can be said that α is a subtype of β if $\varnothing \vdash \alpha \leq ß$ can be derived using the subtyping rules and the equality rules.

(S.1)	$\alpha = \beta \Rightarrow \Gamma \vdash \alpha \leq \beta$
(S.2)	$\Gamma \vdash \alpha \leq \beta, \Gamma \vdash \beta \leq \gamma \Rightarrow \Gamma \vdash \alpha \leq \gamma$
(S.3)	$t \leq s \in \Gamma \Rightarrow \Gamma \vdash t \leq s$
(S.4)	$\Gamma \vdash \bot \leq \alpha$
(S.5)	$\Gamma \vdash \alpha \leq \top$
(S.6)	$\Gamma \vdash \alpha_2 \leq \alpha_1, \Gamma \vdash \beta_1 \leq \beta_2 \Rightarrow \Gamma \vdash \alpha_1 \to \beta_1 \leq \alpha_2 \to \beta_2 \leq$
(S.7)	$\forall \iota\ \varepsilon\ \{1, ..., n\}, \Gamma \vdash \alpha_i \leq \beta_i \Rightarrow \Gamma \vdash \langle a_1 : \alpha_1, ..., a_m : \alpha_m \rangle \leq \langle a_1 : \beta_1, ... ,a_n : \beta_n \rangle$ *with* $n \leq m$
(S.8)	$\forall \iota\ \varepsilon\ \{1, ..., n\}, \Gamma \vdash \alpha_i \leq \beta_i \Rightarrow \Gamma \vdash [a_1 : \alpha_1, ..., a_n : \alpha_n] \leq [a_1 : \beta_1, ... , a_n : \beta_n]$ *with* $n \leq m$
(S.9)	$\forall \iota\ \varepsilon\ \{1, ..., n\}, \Gamma \vdash \alpha_i \leq \beta_i \Rightarrow \Gamma \vdash \alpha_1 \times ... \times \alpha_n \leq \beta_1 \times ... \times \beta_n$
(S.10)	$\Gamma \cup \{t \leq s\} \vdash \alpha \leq \beta \Rightarrow \Gamma \vdash \mu t.\alpha \leq \mu s.\beta$ *with t only in* α*; s only in* β*, t, s not in* Γ.

Figure 3 - Subtyping rules

A.2.2 Type definitions

Elements of *Type* are defined by sets of mutually dependent equations, modelled as well-formed environments. An environment is a finite mapping between type variables and types belonging to *T*, where *T* is the non-recursive subset of *Type*. A well-formed environment Υ is an environment such that the free variables of a type α associated with a variable *t* in the domain of Υ all belong to the domain of Υ. Intuitively, each type variable in an environment represents a type. The associations between type variables and elements of *T* in an environment can be understood as mutually dependent defining equations for the corresponding types.

Formally, let Υ_{wf} be the set of well-formed environments and define $f : A \to_f B$ as a partial function from *A* to *B* with finite domain; $FV(\alpha)$ denotes the set of free variables occurring in α):

$$\Upsilon_{wf} =_{def} \{\Upsilon : T_{var} \to_f Type \mid \forall\ t, t' \in\ dom\ (\Upsilon), t' \in\ FV\ (\Upsilon\ (t)) \Rightarrow t' \in dom\ (\Upsilon)\}$$

Let $\Upsilon = \{t \mapsto \alpha, t1 \mapsto \alpha_1, ..., t_q \mapsto \alpha_q\}$. $\Upsilon \setminus t$ denotes the following environment:

$$\Upsilon \setminus t =_{def} \{t_1 \mapsto \alpha_1 ,...,t_q \mapsto \alpha_q\}$$

The type associated with a type variable *t* in the context of a well-formed environment Υ ($t \in\ dom\ (\Upsilon)$) is defined to be *Val (*t, Υ), where *Val* is the function on types and environments defined recursively in figure 4. Thus any element of *Type* can be defined as *Val (t,* Υ), where Υ is a well-formed environment and $t \in\ dom\ (\Upsilon)$.

A.2.3 An algorithm for type checking

This clause defines an algorithm for type checking which is sound and complete with respect to the type equality and subtyping rules above. The algorithm involves two well-formed environments ε_1 and ε_2 such that $dom(\varepsilon_1) \cap dom(\varepsilon_2) = \varnothing$ (the types to be compared are associated with two variables one in ε_1 and the other in ε_2). It is described as a set of inference rules involving $\varepsilon =_{def} \varepsilon_1 \cup \varepsilon_2$ and a set Σ of the form $\{t_1 \leq s_1, ..., t_n \leq s_n\}$ which records inclusion of variables discovered during execution of the algorithm. An inference rule corresponds to a logical implication of *judgements* of the form

(IT.1)	$Val\,(\bot, \Upsilon) = \bot$
(IT.2)	$Val\,(\top, \Upsilon) = \top$
(IT.3)	$Val\,(Nil, \Upsilon) = Nil$
(IT.4)	$Val\,(\alpha \rightarrow \beta, \Upsilon) = Val\,(\alpha, \Upsilon) \rightarrow Val\,(\beta, \Upsilon)$
(IT.5)	$Val\,(\langle a_1 : \alpha_1, ..., a_n : \alpha_n\rangle, \Upsilon) = \langle a_1 : Val\,(\alpha_1, \Upsilon), ... ,a_n : Val\,(\alpha_n, \Upsilon)\rangle$
(IT.6)	$Val\,([a_1 : \alpha_1, ..., a_n : \alpha_n], \Upsilon) = [a_1 : Val\,(\alpha_1, \Upsilon), ..., a_n : Val\,(\alpha_1, \Upsilon)]$
(IT.7)	$Val\,(\alpha_1 \times ... \times \alpha_n, \Upsilon) = Val\,(\alpha_1, \Upsilon) \times Val\,(\alpha_n, \Upsilon)$
(IT.8)	$if\ t \notin dom\,(\Upsilon)\ then\ Val\,(t, \Upsilon) = t$
(IT.10)	$if\ t \in dom\,(\Upsilon)\ then\ Val\,(t, \Upsilon) = \mu t.Val\,(\Upsilon\,(t), \Upsilon \setminus t)$

Figure 4 - Semantics of interface type definitions

$\Sigma, \varepsilon \vdash \alpha \leq \beta$. A judgement intuitively captures the assertion $\alpha \leq \beta$ holds in the context of Σ and ε. Initial judgements $\{t_1 \leq s_1, ..., t_n \leq s_n\}$ must be such that $\{t_1, s_1, ..., t_n, s_n\} \cap dom\,(\varepsilon) = \emptyset$

The inference rules are given in figure 5. In figure 5 $\alpha, \beta \in Type$, t, s denote arbitrary variables, u denotes variables not in $dom(\varepsilon)$.

Given an initial goal $\Sigma, \varepsilon \vdash \alpha \leq \beta$, the algorithm consists in applying the inference rules backwards, generating subgoals in cases (rec), (fun), (rcd), (pro) and (uni). A tree of goals built in this way is called an execution tree. An execution tree is always finite: if $t \leq s$ is an assumption that is added to Σ, then t and s are type variables in $dom\,(\varepsilon)$; also, the rules (fun), (pro), (uni) and (rcd) reduce the size of the current goal by replacing it with subexpressions of the goal, and each application of (rec) enlarges Σ.

An execution tree *succeeds* if all the leaves correspond to an application of one of the rules (assmp), (bot), (top) or (var). It *fails* if at least one leaf is an unfulfilled goal (i.e. if no rule can be applied to it). If the execution tree corresponding to the goal $\emptyset\varepsilon \vdash \alpha \leq \beta$ succeeds, this is noted $\vdash_A \alpha \leq \beta$.

Given recursive types $\alpha$and β, such that $\alpha = Val\,(t_1, \varepsilon_1)$ and $\beta = Val\,(t_2, \varepsilon_2)$ (ε_1 and ε_2 as above) a subtyping relation, $\leq_A$, is induced by the algorithm by the following definition: $\alpha \leq_A \beta \Leftrightarrow \vdash_A t_1 \leq t_2$.

(assmp)	$t \leq s \in \Sigma \Rightarrow \Sigma, \varepsilon \vdash t \leq s$
(bot)	$\Sigma, \varepsilon \vdash \bot \leq \beta$
(top)	$\Sigma, \varepsilon \vdash a \leq \top$
(var)	$\Sigma, \varepsilon \vdash u \leq u$
(fun)	$\Sigma, \varepsilon \vdash \alpha_2 \leq \alpha_1, \Sigma, \varepsilon \vdash \beta_1 \leq \beta_2 \Rightarrow \Sigma, \varepsilon \vdash \alpha_1 \rightarrow \beta_1 \leq \alpha_2 \rightarrow \beta_2 \leq$
(rcd)	$\forall \iota \in \{1, ..., n\}, \Sigma, \varepsilon \vdash \alpha_i \leq \beta_i \Rightarrow \Sigma, \varepsilon \vdash \langle a_1 : \alpha_1, ..., a_n : \alpha_n\rangle \leq \langle a_1 : \beta_1, ... ,a_n : \beta_n\rangle$ *with* $n \leq m$
(uni)	$\forall \iota \in \{1, ..., n\}, \Sigma, \varepsilon \vdash \alpha_i \leq \beta_i \Rightarrow \Sigma, \varepsilon \vdash [a_1 : \alpha_1, ..., a_n : \alpha_n] \leq \{a_1 : \beta_1, ... , a_m : \beta_m]$ *with* $n \leq m$
(pro)	$\forall \iota \in \{1, ..., n\}, \Sigma, \varepsilon \vdash \alpha_i \leq \beta_i \Rightarrow \Sigma, \varepsilon \vdash \alpha_1 \times ... \times \alpha_n \leq \beta_1 \times ... \times \beta_n$
(rec)	$\Sigma \cup \{t \leq s\}, \varepsilon \vdash \varepsilon\,(t) \leq \varepsilon\,(s) \Rightarrow \Sigma, \varepsilon \vdash t \leq s$

Figure 5 - Type-checking inference rules

This new subtyping relation coincides with the previous one, i.e.,the algorithm is sound and complete with respect to the type equality and type subtyping rules:

- — given α, β in T, if $\alpha \leq_A \beta$ then $\alpha \leq_R \beta$;
- — given α, β in T, if $\alpha \leq_R \beta$ then $\alpha \leq_A \beta$.

A.3 Signal interface signature types

Signal interface signature types are formalized by interpreting them in the *Type* language. The set of signal interface signature types is denotenoted $Type_{sig}$. Elements of $Type_{sig}$ are defined abstractly through the use of two functions: *intype* : $Type_{sig} \rightarrow Type$ and *outype* : $Type_{sig} \rightarrow Type$. In a given signal interface signature type, *intype* describes the set of initiating signals, and *outype* describes the set of responding signals.

Elements of *Type* associated with a signal interface signature type through the *intype* and *outype* functions are defined by well-formed environments with codomain the subset of *Type* defined by the grammar in figure 6, where labels a_i, $i \in \{1, ..., q\}$, are supposed to be distinct. In effect,figure 6 provides an abstract syntax for signal interface signatures. Labels a_i correspond to signal names. *Arg* productions correspond to signal parameters. *Sigsig* productions correspond to individual signal signatures. The functional form adopted for individual signal signature highlights the analogy with announcement signatures.

$$\begin{array}{rcl}
\alpha & ::= & \langle a_i : Sigsig, ..., a_q : Sigsig \rangle \\
Sigssig & ::= & Arg \rightarrow Nil \\
Arg & ::= & Nil \mid t_1 \times ... \times t_p
\end{array}$$

Figure 6 - Abstract syntax for signal interface signature types

The subtyping relation on signal interface types, $\leq_s$, is defined by:

$\forall\, \iota_1, \iota_2 \in Type_{sig}, \iota_1 \leq \iota_2 \equiv \iota_1.intype \leq \iota_2.intype \wedge i_2.outype \leq \iota_1.outype.$

A.4 Operation interface signature types

Operational *server* interface signature types are formalized by interpreting them in the *Type* language (operation client interface signature types can be derived immediately by complementation). The set of operational server interface types is noted $Type_o\,(S)$. Elements of $Type_o\,(S)$ are defined abstractly through the use of the function *optype*: $Type_o\,(S) \rightarrow Type$.

Elements of *Type* associated with an operation interface signature type through the *optype* function are defined by well-formed environments with codomain the subset of *Type* defined by the grammar in figure 7, where labels a_i, $i \in \{1, ..., q\}$, are supposed to be distinct, and where labels c_i, $i \in \{1, ..., q\}$ are supposed to be distinct in the context of an *Opsig* production.

$$\begin{array}{rcl}
\alpha & ::= & \langle a_i : Opsig, ..., a_q : Opsig \rangle \\
Opsig & ::= & Arg \rightarrow Term \mid Arg \rightarrow Nil \\
Term & ::= & [c_1 : Arg, ..., c_q : Arg] \\
Arg & ::= & Nil \mid t_1 \times ... \times t_p
\end{array}$$

Figure 7 - Abstract syntax for operation interface signature types

In effect, figure 7 provides an abstract syntax for operation interface signatures. *Opsig* productions correspond to individual operation signatures. Specifically, *Opsig* productions of the form $Arg \rightarrow Term$ in figure 1 correspond to interrogations.

Opsig productions of the form $Arg \rightarrow Nil$ correspond to announcements. *Arg* productions correspond to invocation parameters. *Term* productions correspond to terminations. *Nil* on the left hand side of an *Opsig* production means that the given invocation does not have any parameter. *Nil* on the right hand side of an *Opsig* production (i.e, in an announcement signature) means that no termination is expected. Labels a_i correspond to operation names. Labels c_1 correspond to termination names.

The subtyping relation on server operational interface types, $\leq_o$, is defined by:

$$\forall \iota_1, \iota_2 \in Type_o\,(S), \iota_1 \leq \iota_2 \equiv \iota_1.optype \leq \iota_2.optype.$$

A.5 Stream interface types

Defining complete signature subtyping rules for stream interfaces is beyond the scope of this Reference Model (see 7.2.4.2). Note however that an individual flow signature type can be formalized by interpreting it in the *Typ*e language. Elements of *Type* associated with a flow signature can be defined by well-formed environments with codomain the subset of *Type* defined by the grammar in figure 8, where label a_i corresponds to the flow name.

α	::=	$\langle a_i : Flowsig \rangle$
Flowsig	::=	$Arg \rightarrow Nil$
Arg	::=	$Nil \mid t$

Figure 8 - Abstract syntax for stream interface signature types

The subtyping rule in clause 7.2.4.2 associated with corresponding flows (assuming they have the same causality) just corresponds to the subtype relation, ≤, in this case.

A.6 Example

Consider the following server operation interface signature type definitions (i.e., the well-formed environment)

$$\Upsilon = \{t \mapsto \alpha, f_t \mapsto \beta\}$$

where

$$\alpha =_{def} \quad \langle\, op : t \rightarrow [ok : Nil, nok : Nil], factory : Nil \rightarrow [ok : f_t]\,\rangle$$

$$\beta =_{def} \quad \langle\, new : Nil \rightarrow [ok : t]\,\rangle$$

and where $t, f_t \in Tvar$; *op*, *factory*, *new*, *ok* and *nok* $\in \Lambda$ (*op*, *factory* and *new* are operation names; *ok* and *nok* are termination names).

Intuitively, the environment Υ corresponds to the definition of two types, *t* and f_t. f_t is equipped with only one operation *new*, that takes no argument and returns a reference to an instance of type *t*. One may imagine, for instance, that f_t is the type of a factory object that creates objects with an interface of type *t* on demand, i.e.,on each invocation of operation new. *t* is equipped with two operations: *op* and *factory*. Operation *op* takes a reference to an instance of type *t* as argument: this is a first instance of a recursive definition. Operation *factory* takes no argument and returns a reference to an instance of type f_t. One may imagine for example that, for management purposes, each object with an interface of type *t* is able, on request (i.e., upon invocation of operation *factory*), to return a reference to the factory that created it. This is a second instance of a recursive definition, since the definition of f_t refers to *t*.

Applying the definition of *Val* given above, defining $\Upsilon_1 =_{def} \{ f_t \mapsto \beta\}$, overloading the = sign and using type equivalence rule E.10 the following is obtained:

$$\begin{aligned} Val\,(t, \Upsilon) &= \mu\, t.\, Val\,(\alpha, \{ f_t \mapsto \beta \} \\ &= \mu\, t.\, \langle op : Val(t, \Upsilon_1) \rightarrow [ok: Nil, nok : Nil], \\ &\quad factory: Nil \rightarrow [ok : Val\,(f_t, \Upsilon_1)]\rangle \\ &= \mu\, t.\, \langle op : t \rightarrow [ok: Nil, nok : Nil], \end{aligned}$$

$factory: Nil \rightarrow [ok : \mu\, f_t.Val\, (\beta, \emptyset)]\rangle$
$= \mu\, t.\, \langle op : t \rightarrow [ok: Nil, nok : Nil],$
$factory: Nil \rightarrow [ok : \mu\, f_t.\langle new : Nil \rightarrow [ok : t]\rangle]\rangle$
$= \mu\, t.\, \langle op : t \rightarrow [ok: Nil, nok : Nil],$
$factory: Nil \rightarrow [\mathrm{ok} : \langle new : Nil \rightarrow [ok : \mathrm{t}]\rangle]\rangle.$