Non-regular Process Types

Franz Puntigam

Technische Universität Wien, Institut für Computersprachen
Argentinierstraße 8, A-1040 Vienna, Austria
`franz@complang.tuwien.ac.at`

Abstract. Process types specify sequences of acceptable messages. Even if the set of acceptable messages changes dynamically, a type checker can statically ensure that only acceptable messages are sent. As proposed so far, all message sequence sets specified by types can be generated by regular grammars. We propose to increase the expressiveness so that non-regular message sequence sets can be specified. Type equivalence and subtyping take possible type extensions into account.
Keywords: type systems, subtyping, active objects

1 Introduction

A process type [13, 14, 15] specifies not only a set of messages, but also constraints on acceptable sequences of these messages. Type safety is checked statically by ensuring that each object reference is associated with an appropriate "type mark"; it specifies all message sequences that can be sent through the reference.

An instance of a subtype can be used wherever an instance of a supertype is expected [6]. In a process type system, a subtype can extend a supertype by specifying additional messages and less constraints on message sequences. Messages accepted by objects of supertypes are also accepted by objects of subtypes.

It is decidable if two regular grammars generate the same language and if the language generated by a regular grammar is contained in that generated by another [5]. These relations are undecidable (or not known to be decidable) for more expressive grammars like LR(1) and context-free grammars. Process types shall be extended so that non-regular languages can be expressed. Since equivalence and containment of message sequence sets are undecidable, stricter notions of type equivalence and subtyping shall be used.

According to Liskov [6], a type is a partial specification of object behavior. A subtype specifies the behavior in more detail. If a supertype allows a user to rely on some property, a subtype has to allow the user to rely on the same property. For example, if a supertype always allows a user to send a message "put", a subtype must not allow the user to send "delete" if "delete" is the last acceptable message. This restriction ensures that several users can safely send messages to the object, although each user knows a different type.

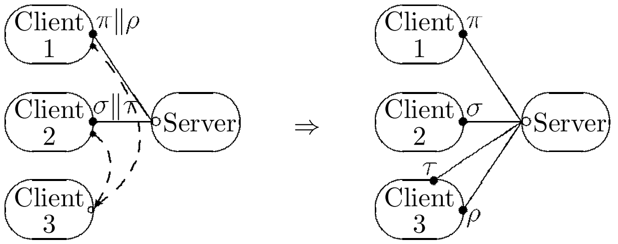
Process types enforce this restriction in a different way: If a user can send "delete", a type splitting rule ensures that no other user can send "put". The set of acceptable message sequences does not reflect the restriction. There is no loss in flexibility if the restriction is enforced by the type equivalence and subtyping relations. We shall show that such relations are sound and complete.

Syntax and informal semantics of non-regular process types are introduced in Section 2. The type equivalence and subtyping relations are defined in Section 3. A characterization of these relations concerning acceptable message sequences is given in Section 4. A discussion of related work follows in Section 5.

2 Specification of Non-regular Process Types

A process type (as presented in [14, 15]) consists of two parts – a set of message descriptors and a state. A message descriptor specifies the signature of a message, conditions for accepting such messages depending on the state, and updates of the state on message acceptance. States are represented by multi-sets of tokens, and conditions by the availability of tokens in states. The state of an object's type is updated when a message is accepted, that of the type associated with a reference (a type mark) when a message is sent through the reference. The tokens in the type marks' states are, essentially, contained in the state of the object's type. This concept works fine for regular process types, where all conditions correspond to the availability of a minimum number of tokens.

We must deal with aliasing: Let a server accept messages according to a type $\pi\|\rho\|\sigma\|\tau$; at least the sequences specified by $\pi, \dots, \tau$ are accepted in arbitrary interleaving. Two clients have references to the server with the type marks $\pi\|\rho$ and $\sigma\|\tau$, respectively. If both clients send a message containing a reference to the server as argument to a further object, the type marks must be split:



The type marks π and σ remain at the references in the first two clients, while Client 3 gets two new references with the type marks τ and ρ. Now, messages can be sent to the server through four independent references.
Static type checking is possible: The compiler has to ensure that (1) all message sequences specified by the objects' types are accepted by the objects, (2) only messages specified by the references' type marks are sent through the references, and (3) type marks are split when an object reference is used as argument. Parameter types in message signatures determine how to split types.

In non-regular process types, the acceptability of messages may depend on the exact number of available tokens of some kind. Such conditions can be used only if all tokens of a kind occur in a single type mark's state. States shall hold information about whether other type marks may contain further tokens.

Type representation. An infinite set of names (denoted by $u, v, w, \dots$) and a set of constant symbols ($x, y, z, \dots$) are considered given. Names are used as parameters, symbols as message selectors and tokens. Parameters are underlined when occurring at positions where types or integers shall be substituted for them. An underlined occurrence binds all following free occurrences. Some conventions simplify the notation: For each meta-symbol e, $\tilde{e}$ is an abbreviation of $e_1, \dots, e_n$ for arbitrary n. For example, $\tilde{u}$ denotes a list of names $u_1, \dots, u_n$. Likewise, $\{\tilde{e}\}$ denotes the set consisting of $e_1, \dots, e_n$, and $|\tilde{e}|$ the length of the sequence. For a binary operator $\circ$, $\tilde{e} \circ \tilde{g}$ stands for $e_1 \circ g_1, \dots, e_n \circ g_n$ ($|\tilde{e}| = |\tilde{g}|$). All names in an underlined parameter list $\underline{\tilde{u}}$ are regarded as pairwise different.

This is the syntax of non-regular process types (denoted by $\pi, \rho, \sigma, \tau, \dots$):

$$
\begin{array}{rcl@{\qquad}rcl}
\tau & ::= & \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{r}] \mid *\underline{u}\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{r}] \mid u \mid \tau_1\|\tau_2 & \mu & ::= & \text{type} \mid \text{int} \\
\alpha & ::= & a[\tilde{r}_1][\tilde{r}_2] & r & ::= & x^p \mid x^{p_1|p_2} \\
a & ::= & x\langle\underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\tau}\rangle & p & ::= & n \mid \infty \mid u \mid p_1 + p_2
\end{array}
$$

A type $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{r}]$ consists of a set of message descriptors $\tilde{\alpha}$ and a state $[\tilde{r}]$. Each state descriptor ($r, s, t, \dots$) is of the form x^p or $x^{p|q}$, where x is a symbol used as token, p a multiplication factor and q a splitting factor. $[\tilde{r}]$ is regarded as a multi-set containing p tokens x for each x^p or $x^{p|q}$ in $\tilde{r}$; x^1 is abbreviated by x. State descriptors $x^{p|q}$ specify that message sequences may depend on the exact number of tokens x. The state has been split q times. If a state contains $x^{p_1|q_1}, \dots, x^{p_n|q_n}$, the type has about $1/2^{q_1} + \dots + 1/2^{q_n}$ of the tokens x to be considered. The type has all available tokens of symbol x if $x^{p|0}$ is in the state.

Integers ($p, q, \dots$) are nonnegative integer constants, ∞ denoting a very large (infinite) integer, parameters standing for integers, and integer addition.

Message descriptors ($\alpha, \beta, \gamma, \dots$) are of the form $a[\tilde{s}][\tilde{t}]$, where a is a message signature, $[\tilde{s}]$ an in-set and $[\tilde{t}]$ an out-set. A message of signature a is acceptable if each state descriptor in the in-set is in the type's state $[\tilde{r}]$ and, for each state descriptor $x^{p|0}$ in the in-set, no further state descriptor for token x is in $[\tilde{r}]$; the type is updated by removing all state descriptors in the in-set from the state and adding those in the out-set. A state descriptor $x^{p|q}$ can occur in an in-set only with $q = 0$, and $x^{p|0}$ in an out-set only if an $x^{q|0}$ is in the in-set.

A message signature ($a, b, c, \dots$) is of the form $x\langle\underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\tau}\rangle$, where x is the message selector, the $\tilde{u}$ are type and integer parameters, the $\tilde{\mu}$ meta-types and the $\tilde{\tau}$ types of object parameters; the $\tilde{u}$ can occur in the $\tilde{\tau}$. Meta-types ($\mu, \nu, \dots$) are "type", the type of all type expressions, and "int", the type of all integers.

A type $*\underline{u}\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{r}]$ is recursive: The type parameter u can occur in the message signatures in $\tilde{\alpha}$. Type parameters can occur wherever types can occur. A type $\sigma\|\tau$ denotes the combination of two types; it can be split into σ and τ.

Examples show static parts of types for simple data stores:

$$
\begin{array}{rl}
\mathit{DS} \overset{\text{def}}{=} & \boldsymbol{\{}\text{put}\langle\underline{u}{:}\text{type}, u\rangle[\text{empty}][\text{full}],\ \text{get}\langle\mathit{Back}[\text{one}]\rangle[\text{full}][\text{empty}]\boldsymbol{\}} \\
\mathit{Back} \overset{\text{def}}{=} & \boldsymbol{\{}\text{back}\langle\underline{u}{:}\text{type}, u\rangle[\text{one}][]\boldsymbol{\}} \\
\mathit{DS}' \overset{\text{def}}{=} & \boldsymbol{\{}\text{put}\langle\underline{u}{:}\text{type}, u\rangle[][\text{full}],\ \text{get}\langle\mathit{Back}[\text{one}]\rangle[\text{full}][]\boldsymbol{\}} \\
\mathit{DS}'' \overset{\text{def}}{=} & \boldsymbol{\{}\text{put}\langle\underline{u}{:}\text{type}, u\rangle[\text{ok}][\text{full}],\ \text{get}\langle\mathit{Back}[\text{one}]\rangle[\text{ok}, \text{full}][],\ \text{delete}\langle\rangle[\text{ok}^{\infty|0}][]\boldsymbol{\}}
\end{array}
$$

An object of type $DS[\text{empty}^{50}]$ accepts a message "put" whenever one of the fifty buffer slots is empty, and "get" if a slot is full; the empty slot becomes full, and the full slot empty. The type $DS[\text{empty}^{\infty}]$ is, in some sense, equivalent to $DS'[]$: An infinite data store of this type always accepts "put", but "get" only when there is a full slot. Instances of $DS'[\text{full}^{\infty}]$ accept "put" and "get" in arbitrary ordering. A data store of the type $DS''[\text{ok}^{\infty|0}]$ allows users to explicitly delete the store by sending "delete". An infinite number of tokens "ok" is present in the state as long as the data store is alive; the tokens are removed when "delete" is accepted. Because of $\text{ok}^{\infty|0}$ in the in-set, "delete" can be sent only if there is no other reference to the data store with a token "ok" in the type mark's state.

The expressiveness can be shown by a further example: An object of type

$$\begin{array}{l}\{a_1[x^{1|0}][x^{1|0},y],\ a_2[x^{1|0},y][x^{2|0},y,z],\ a_2[x^{2|0}][x^{2|0},z],\\ \quad a_3[x^{2|0},y][x^{3|0}],\ a_3[x^{3|0},y][x^{3|0}],\ a_4[x^{3|0},y^{0|0},z][x^{3|0},y^{0|0}]\}[x^{1|0},y^{0|0}]\end{array}$$

accepts messages sequences of the form $a_1^n a_2^m a_3^n a_4^m$ (with $m, n > 0$). Using context-free grammars it is not possible to specify words of this form.

3 Type Equivalence and Subtyping

Some further notation is needed in the definition of the type equivalence and subtyping relations. $\mathit{free}(\tilde{e})$ denotes the set of all names occurring free in at least one expression in $\tilde{e}$. $\mathit{tok}(\tilde{r})$ denotes the set $\{x \mid \exists_{p\neq 0}\, x^p \in \{\tilde{r}\} \vee \exists_{p,q}\, x^{p|q} \in \{\tilde{r}\}\}$. $[\tilde{e}/\tilde{u}]\tilde{g}$ denotes the simultaneous substitution of e_i for all free occurrences of u_i ($i = 1, \ldots, |\tilde{e}|$) in $\tilde{g}$, where $|\tilde{e}| = |\tilde{u}|$ and all names $\tilde{u}$ are pairwise different.

A symbol x is relevant as token for $\tilde{\alpha}$ if x^p (with $p \neq 0$) or $x^{p|q}$ occurs in the in-set of at least one message descriptor in $\tilde{\alpha}$; $\mathit{relev}(\tilde{\alpha})$ denotes the set of all symbols relevant for $\tilde{\alpha}$: $\mathit{relev}(a_1[\tilde{r}_1][\tilde{s}_1], \ldots, a_n[\tilde{r}_n][\tilde{s}_n]) = \mathit{tok}(\tilde{r}_1, \ldots, \tilde{r}_n)$.

The exact number of available tokens x is known for $\{\tilde{\alpha}\}[\tilde{r}]$ if $x \in \mathit{relev}(\tilde{\alpha})$ and an $x^{p|q}$ occurs in $[\tilde{r}]$; $\mathit{exact}(\{\tilde{\alpha}\}[\tilde{r}])$ denotes the set of all such symbols.

The state descriptor removing relation $\xrightarrow{\text{rem } x}$ on message descriptors is defined by $a[\tilde{r}][\tilde{s}] \xrightarrow{\text{rem } x} a[\tilde{r}'][\tilde{s}']$, where $[\tilde{r}] = [\tilde{r}']$ if there is an $x^{p|0} \in \{\tilde{r}\}$, and $[\tilde{s}] = [\tilde{s}']$ if there is an $x^{p|0} \in \{\tilde{s}\}$; otherwise $[\tilde{r}']$ and $[\tilde{s}']$ are constructed by removing all state descriptors x^q from $[\tilde{r}]$ and $[\tilde{s}]$, respectively.

A subtyping environment contains subtyping assumptions of the form $u \leq v$.

Structural equivalence $\equiv$ on types and their constituents is the least congruence closed under renaming of bound names (α-conversion) and these rules:

$$\begin{array}{rcc}(p + p') + q \equiv p + (p' + q) & p + q \equiv q + p & p + 0 \equiv p \\ p + p' \equiv q\ (p + p' = q) & p + \infty \equiv \infty & [\tilde{r}, t, \tilde{s}] \equiv [\tilde{r}, \tilde{s}, t] \\ [\tilde{r}, x^{p_1|q+1}, x^{p_2|q+1}] \equiv [\tilde{r}, x^{p_1+p_2|q}] & [\tilde{r}, x^0] \equiv [\tilde{r}] & [\tilde{r}, x^p, x^q] \equiv [\tilde{r}, x^{p+q}] \\ [\tilde{r}, x^{p_1}, x^{p_2|q}] \equiv [\tilde{r}, x^{p_1+p_2|q}] & \{\tilde{\alpha}, \beta, \beta\} \equiv \{\tilde{\alpha}, \beta\} & \{\tilde{\alpha}, \beta, \tilde{\gamma}\} \equiv \{\tilde{\alpha}, \tilde{\gamma}, \beta\}\end{array}$$

Definition 1. *Type equivalence $\Pi \vdash \sigma \cong \tau$ (or $\sigma \cong \tau$ for empty Π) is the least reflexive, transitive and symmetric relation closed under these rules:*

$$\Pi \vdash \sigma \cong \tau \quad (\sigma \equiv \tau) \qquad (\mathrm{equiv}_{\cong})$$

$$\Pi \vdash (\rho \| \sigma) \| \tau \cong \rho \| (\sigma \| \tau) \qquad (\mathrm{assoc}_{\cong})$$

$$\Pi \vdash \sigma \| \tau \cong \tau \| \sigma \qquad (\mathrm{commut}_{\cong})$$

$$\Pi \vdash \tau \| \{\}[] \cong \tau \qquad (\mathrm{empty}_{\cong})$$

$$\Pi \vdash *\underline{u}\{\tilde{\alpha}\}[\tilde{r}] \cong \{[*\underline{u}\{\tilde{\alpha}\}[\tilde{r}]/u]\tilde{\alpha}\}[\tilde{r}] \qquad (\mathrm{rec}_{\cong})$$

$$\Pi \vdash \{\tilde{\alpha}\}[\tilde{r}] \| \{\tilde{\gamma}\}[\tilde{s}] \cong \{\tilde{\alpha}, \tilde{\gamma}\}[\tilde{r}, \tilde{s}] \quad (1) \qquad (\mathrm{comb}_{\cong})$$

$$\Pi \vdash *\underline{u}\{\tilde{\alpha}\}[\tilde{r}, s] \cong *\underline{u}\{\tilde{\alpha}\}[\tilde{r}] \quad (\mathit{tok}(s) \cap \mathit{relev}(\tilde{\alpha}) = \emptyset) \qquad (\mathrm{state}_{\cong})$$

$$\Pi \vdash *\underline{u}\{\tilde{\alpha}, \beta\}[\tilde{r}] \cong *\underline{u}\{\tilde{\alpha}\}[\tilde{r}] \quad (\beta{=}a[\tilde{s}, x^{p|0}][\tilde{t}]; x{\notin}\mathit{exact}(\{\tilde{\alpha}\}[\tilde{r}])) \qquad (\mathrm{descr}_{\cong})$$

$$\Pi \vdash *\underline{u}\{\tilde{\alpha}\}[\tilde{r}, x^{\infty}] \cong *\underline{u}\{\tilde{\gamma}\}[\tilde{r}] \quad (x{\in}\mathit{relev}(\tilde{\alpha}); x{\notin}\mathit{tok}(\tilde{r}); \tilde{\alpha} \xrightarrow{\mathit{rem}\ x} \tilde{\gamma}) \qquad (\mathrm{avail}_{\cong})$$

$$\Pi \vdash *\underline{u}\{\tilde{\alpha}, \tilde{\gamma}\}[\tilde{r}] \cong *\underline{u}\{\tilde{\alpha}\}[\tilde{r}] \quad (2) \qquad (\mathrm{red}_{\cong})$$

(1) $\mathit{tok}(\tilde{r}) \subseteq \mathit{relev}(\tilde{\alpha});\ \forall_x (x \in \mathit{relev}(\tilde{\gamma}) \wedge \exists_{p,q}\, x^{p|q} \in \{\tilde{r}\}) \Rightarrow \exists_{p,q}\, x^{p|q} \in \{\tilde{s}\};$
$\mathit{tok}(\tilde{s}) \subseteq \mathit{relev}(\tilde{\gamma});\ \forall_x (x \in \mathit{relev}(\tilde{\alpha}) \wedge \exists_{p,q}\, x^{p,q} \in \{\tilde{s}\}) \Rightarrow \exists_{p,q}\, x^{p,q} \in \{\tilde{r}\}$

(2) $\forall_{\gamma \in \{\tilde{\gamma}\}} \exists_{\alpha \in \{\tilde{\alpha}\}}\, \alpha \succeq \gamma,$ *where* $x\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\sigma}\rangle[\tilde{s}][\tilde{t}, \tilde{t}'] \succeq x\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\tau}\rangle[\tilde{s}, \tilde{s}', \tilde{s}''][\tilde{s}'', \tilde{t}, \tilde{t}'']$
if $\Pi \vdash \tilde{\tau} \leq \tilde{\sigma}$ *and* $\mathit{tok}(\tilde{s}', \tilde{t}', \tilde{t}'') \cap \mathit{exact}(\{\tilde{\alpha}, \tilde{\gamma}\}[\tilde{r}]) = \emptyset = \mathit{tok}(\tilde{t}'') \cap \mathit{relev}(\tilde{\alpha})$

Type combination is an associative, commutative operation with $\{\}[]$ as neutral element. Two types $\{\tilde{\alpha}\}[\tilde{r}]$ and $\{\tilde{\gamma}\}[\tilde{s}]$ are combined to $\{\tilde{\alpha}, \tilde{\gamma}\}[\tilde{r}, \tilde{s}]$ as specified by (comb$_{\cong}$). The side-conditions ensure that (1) all symbols irrelevant as tokens have been removed by applying (state$_{\cong}$), and (2) if a state descriptor $x^{p|q}$ occurs in $[\tilde{r}, \tilde{s}]$, state descriptors of this form occur in both $[\tilde{r}]$ and $[\tilde{s}]$ if x is relevant there.

Rule (descr$_{\cong}$) removes message descriptors if the acceptability of such messages depends on an exact number of tokens, but this number is not known.

State descriptors that specify always available tokens can be removed from states, in-sets and out-sets simultaneously by using (avail$_{\cong}$).

Rule (red$_{\cong}$) allows us to remove redundant message descriptors. A message descriptor is redundant if there is another message descriptor that can be used wherever the redundant one can be used.

Definition 2. *Subtyping $\Pi \vdash \sigma \leq \tau$ (or $\sigma \leq \tau$ if Π is empty) is the least reflexive and transitive relation closed under these rules:*

$$\Pi \cup \{u \leq v\} \vdash u \leq v \qquad (\mathrm{assmp}_{\leq})$$

$$\frac{\exists_\rho\, \Pi \vdash \sigma \cong \tau \| \rho}{\Pi \vdash \sigma \leq \tau} \qquad (\mathrm{sub}_{\leq})$$

$$\frac{\Pi \vdash \pi \leq \rho \quad \Pi \vdash \sigma \leq \tau}{\Pi \vdash \pi \| \sigma \leq \rho \| \tau} \qquad (\mathrm{comb}_{\leq})$$

$$\frac{\Pi \cup \{u \leq v\} \vdash \{\tilde{\alpha}\}[\tilde{r}] \leq \{\tilde{\gamma}\}[\tilde{s}]}{\Pi \vdash *\underline{u}\{\tilde{\alpha}\}[\tilde{r}] \leq *\underline{v}\{\tilde{\gamma}\}[\tilde{s}]} \quad (u \notin \mathit{free}(\tilde{\gamma}, \tilde{s}, \Pi);\ v \notin \mathit{free}(\tilde{\alpha}, \tilde{r}, \Pi)) \qquad (\mathrm{rec}_{\leq})$$

Rule ($\text{sub}_{\leq}$) states that σ is a subtype of τ if there is a type ρ such that σ and the combination of τ and ρ are equivalent; ρ specifies the difference between σ and τ. As a special case with $\rho = \mathbf{\{\}}[]$, ($\text{sub}_{\leq}$) states that $\Pi \vdash \sigma \cong \tau$ implies $\Pi \vdash \sigma \leq \tau$. With $\rho = \sigma$, $\Pi \vdash \sigma \leq \mathbf{\{\}}[]$ follows from ($\text{sub}_{\leq}$). Rules like ($\text{rec}_{\leq}$) were shown to be useful as definition of subtyping for recursive types [1].

Type equivalence and subtyping are decidable relations. A proof can be found in an extended version of the present paper [16].

Using the above examples, it is easy to derive $\mathit{DS}[\text{empty}^{\infty}] \cong \mathit{DS}'[]$ as well as $\mathit{DS}[\text{empty}^{10}] \| \mathit{DS}[\text{empty}^{40}] \cong \mathit{DS}[\text{empty}^{50}]$ and $\mathit{DS}[\text{empty}^{50}] \| \mathit{DS}'[] \cong \mathit{DS}'[]$ from the rules. Hence, $\mathit{DS}'[] \leq \mathit{DS}[\text{empty}^{50}]$ and $\mathit{DS}[\text{empty}^{50}] \leq \mathit{DS}[\text{empty}^{10}]$ hold. However, $\mathit{DS}''[\text{ok}^{\infty|0}] \leq \mathit{DS}'[]$ does not hold. But, $\mathit{DS}''[\text{ok}^{\infty|0}]$ is a subtype of the type $\mathbf{\{}\text{put}\langle \underline{u}\text{:type}, u\rangle[\text{ok}][\text{full}],\ \text{get}\langle \mathit{Back}[\text{one}]\rangle[\text{ok}, \text{full}][]\mathbf{\}}[\text{ok}^{\infty|0}]$. Although this type specifies the same set of acceptable message sequences as $\mathit{DS}'[]$ (see below), the types are not equivalent.

4 Message Signature Sequences

Definition 3. *A type $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{r}]$ with $\alpha_i = x_i\langle \tilde{\underline{u}}_i{:}\tilde{\mu}_i, \tilde{\tau}_i\rangle[\tilde{s}_i][\tilde{t}_i]$ is deterministic if for all $i, j = 1..|\tilde{\alpha}|$ $(i \neq j)$ with $x_i = x_j$ and $|\tilde{\mu}_i, \tilde{\tau}_i| = |\tilde{\mu}_j, \tilde{\tau}_j|$ implies one of these:*

- $\exists_{x,p}\,(\exists_{\tilde{t}'}\,[\tilde{s}_i] \equiv [x^{p|0}, \tilde{t}'] \wedge p \not\equiv \infty \wedge \forall_q\, x^q \notin \{\tilde{t}'\}) \Rightarrow \exists_{\tilde{s}'}\,[\tilde{s}_j] \equiv [x^{p+1}, \tilde{s}']$;
- *or* $\exists_{x,p}\,(\exists_{\tilde{t}'}\,[\tilde{s}_j] \equiv [x^{p|0}, \tilde{t}'] \wedge p \not\equiv \infty \wedge \forall_q\, x^q \notin \{\tilde{t}'\}) \Rightarrow \exists_{\tilde{s}'}\,[\tilde{s}_i] \equiv [x^{p+1}, \tilde{s}']$.

The type of each object shall be deterministic to ensure that, when a message is accepted, this type is updated in the same way as the corresponding type mark was updated when the message was sent. Deterministic types are free of redundant message descriptors. All examples of types of the form $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{r}]$ shown above are deterministic. Types used as type marks need not be deterministic, but they are supertypes of deterministic types. Subtyping keeps all important properties of deterministic types. Especially, each type of the form $\sigma\|\tau$ in a type-consistent system is equivalent to a type of the form $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{r}]$. Hence, in the rest of this paper, we consider mainly types of this form.

Definition 4. *A message signature a is active with follow-state $[\tilde{s}]$ in a type $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{r}]$ if $a[\tilde{s}] \in \mathit{act}(\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{r}])$, where act is defined by:*

$$\begin{aligned}\mathit{act}(\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{r}]) = \{a[\tilde{t}, \tilde{t}'] \mid a[\tilde{s}][\tilde{t}] \in \{\tilde{\alpha}\};\ [\tilde{r}] \equiv [\tilde{s}, \tilde{t}'];\ \forall_{x,p}\, x^{p|0} {\in} \{\tilde{s}\} \Rightarrow x {\notin} \mathit{tok}(\tilde{t}');\\ \forall_{x,p}(x^{\infty} {\in} \{\tilde{r}\} \wedge x^{p|0} {\notin} \{\tilde{s}\}) \Rightarrow x^{\infty} {\in} \{\tilde{t}'\}\ \}\end{aligned}$$

If a message signature a is active with a follow-state $[\tilde{s}]$ in a type $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{r}]$, an object of this type accepts a (single) message of signature a. When the message is accepted, the type is updated to $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{s}]$. The signatures of the messages acceptable next are active in $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{s}]$. Type marks are updated in the same way when sending messages. A message descriptor $a[\tilde{s}][\tilde{t}]$ is active in a type $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{r}]$ with follow-state $[\tilde{r}']$ if $a[\tilde{s}][\tilde{t}] \in \{\tilde{\alpha}\}$ and $a[\tilde{r}'] \in \mathit{act}(\mathbf{\{}a[\tilde{s}][\tilde{t}]\mathbf{\}}[\tilde{r}])$.

Definition 5. *The set* $seq(\tau)$ *of all message signature sequences conforming to a type* τ *of the form* $\{\tilde{\alpha}\}[\tilde{r}]$ *is constructed inductively:*

$$\begin{aligned} S_0 &= \{\langle\rangle[\tilde{r}]\} \\ S_{i+1} &= \{\langle\tilde{\alpha}, x\langle\underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\sigma}\rangle\rangle[\tilde{s}] \mid \langle\tilde{\alpha}\rangle[\tilde{t}] \in S_i;\ x\langle\underline{\tilde{v}}{:}\tilde{\mu}, \tilde{\tau}\rangle[\tilde{s}] \in act(\{\tilde{\alpha}\}[\tilde{t}]); \\ &\qquad\qquad \tilde{\sigma} \le [\tilde{u}/\tilde{v}]\tilde{\tau};\ \{\tilde{u}\} \not\subseteq free(\tilde{\tau}) \setminus \{\tilde{v}\}\ \} \qquad (i \ge 0) \\ seq(\tau) &= \{\langle\tilde{\alpha}\rangle \mid \langle\tilde{\alpha}\rangle[\tilde{s}] \in \textstyle\bigcup_{i\ge 0} S_i\} \end{aligned}$$

Theorem 1. *Let* σ *and* τ *be types of the form* $\{\tilde{\alpha}\}[\tilde{r}]$*. Then,* $\sigma \cong \tau$ *implies* $seq(\sigma) = seq(\tau)$*, and* $\sigma \le \tau$ *implies* $seq(\tau) \subseteq seq(\sigma)$*.*

Proof. First, the proof of $seq(\sigma) = seq(\tau)$ if $\sigma \cong \tau$ is sketched. Most type equivalence rules do not influence whether a message signature is active. Only (red$_\cong$) is a bit more difficult. Let σ be of the form $\{\tilde{\alpha}\}[\tilde{r}]$ and τ of the form $\{\tilde{\gamma}\}[\tilde{s}]$; and let (red$_\cong$) be applied to σ and τ such that $[\tilde{r}] \cong [\tilde{s}, \tilde{t}]$ for some $\tilde{t}$ with $tok(t) \cap relev(\tilde{\gamma}) = \emptyset$, and $\{\tilde{\alpha}\} \cong \{\tilde{\gamma}, \tilde{\beta}\}$ for some $\tilde{\beta}$; the message descriptors $\tilde{\beta}$ are removed. Hence, $seq(\tau) \subseteq seq(\sigma)$. A message descriptor is deleted only if there remains a more general message descriptor being active whenever the removed one was active. This implies $seq(\sigma) \subseteq seq(\tau)$ and $seq(\sigma) = seq(\tau)$.

If there is a type $\{\tilde{\alpha}\}[\tilde{r}]$ with $\{\tilde{\alpha}\}[\tilde{r}] \cong \sigma\|\tau$, then $\langle\tilde{a}\rangle \otimes \langle\tilde{c}\rangle \subseteq seq(\{\tilde{\alpha}\}[\tilde{r}])$ for each $\langle\tilde{a}\rangle \in seq(\sigma)$ and $\langle\tilde{c}\rangle \in seq(\tau)$, where $\langle\tilde{a}\rangle \otimes \langle\tilde{c}\rangle$ is the set of all arbitrary interleavings of $\langle\tilde{a}\rangle$ and $\langle\tilde{c}\rangle$. This is easy to see from the definition of *seq*.

Let $\sigma \le \tau$. There is a type ρ with $\sigma \cong \tau\|\rho$ according to (sub$_\le$) and, therefore, $\langle\tilde{a}\rangle \otimes \langle\rangle \subseteq seq(\sigma)$ for all $\langle\tilde{a}\rangle \in seq(\tau)$. Hence, $seq(\tau) \subseteq seq(\sigma)$. □

The reverses of Theorem 1 do not hold: $seq(\sigma) = seq(\tau)$ does not imply $\sigma \cong \tau$, and $seq(\tau) \subseteq seq(\sigma)$ does not imply $\sigma \le \tau$. A more accurate characterization of $\cong$ and $\le$ concerning message signature sequences considers possible extensions of types. If $\sigma \cong \tau$ (or $\sigma \le \tau$) holds, $\sigma\|\rho \cong \tau\|\rho$ (or $\sigma\|\rho \le \tau\|\rho$) are also expected to hold for each type ρ combinable with σ and τ. The next theorems show soundness and completeness of $\cong$ and $\le$: $seq(\sigma\|\rho) = seq(\tau\|\rho)$ (or $seq(\tau\|\rho) \subseteq seq(\sigma\|\rho)$) for all appropriate ρ is equivalent to $\sigma \cong \tau$ (or $\sigma \le \tau$).

Definition 6. *Two types* σ *and* τ *of the form* $\{\tilde{\alpha}\}[\tilde{r}]$ *are extensibility-equivalent (denoted by* $\sigma \simeq \tau$*) if and only if for each type* ρ*:* $seq(\{\tilde{\beta}\}[\tilde{s}]) = seq(\{\tilde{\gamma}\}[\tilde{t}])$ *for all* $\tilde{\beta}$*,* $\tilde{\gamma}$*,* $\tilde{s}$ *and* $\tilde{t}$ *with* $\sigma\|\rho \cong \{\tilde{\beta}\}[\tilde{s}]$ *and* $\tau\|\rho \cong \{\tilde{\gamma}\}[\tilde{t}]$*.*

Theorem 2. *Let* σ *and* τ *be types of the form* $\{\tilde{\alpha}\}[\tilde{r}]$*. Then,* $\sigma \cong \tau \Leftrightarrow \sigma \simeq \tau$*.*

Proof. $\sigma \cong \tau \Rightarrow \sigma \simeq \tau$ follows from Theorem 1. The other direction can be shown by a case analysis on all reasons for $\sigma \not\cong \tau$ if $seq(\sigma) = seq(\tau)$. A type ρ not satisfying the conditions of Def. 6 is constructed for each reason [16]. □

Definition 7. *For two types* σ *and* τ *of the form* $\{\tilde{\alpha}\}[\tilde{r}]$*,* σ *is extensibility-substitutable for* τ *(denoted by* $\sigma \lesssim \tau$*) if and only if for each type* ρ*:* $seq(\{\tilde{\gamma}\}[\tilde{t}]) \subseteq seq(\{\tilde{\beta}\}[\tilde{s}])$ *for all* $\tilde{\beta}$*,* $\tilde{\gamma}$*,* $\tilde{s}$ *and* $\tilde{t}$ *with* $\sigma\|\rho \cong \{\tilde{\beta}\}[\tilde{s}]$ *and* $\tau\|\rho \cong \{\tilde{\gamma}\}[\tilde{t}]$*.*

Theorem 3. *Let σ and τ be types of the form $\{\tilde{\alpha}\}[\tilde{r}]$. Then, $\sigma \leq \tau \Leftrightarrow \sigma \lesssim \tau$.*

Proof. $\sigma \leq \tau \Rightarrow \sigma \lesssim \tau$ follows from Theorem 1. The other direction can be shown by a case analysis on all reasons for $\sigma \not\leq \tau$ if $seq(\tau) \subset seq(\sigma)$. A type ρ not satisfying the conditions of Def. 7 is constructed for each reason [16]. □

5 Related Work

Most work on types in concurrent systems is based on Milner's π-calculus [8, 7] and similar calculi. The problems of inferring most general types [4, 19] and subtyping [11, 12, 18, 3, 17] were considered. But these type systems cannot represent message sequences and ensure statically that all sent messages are acceptable.

A large amount of work based on "path expressions" and process algebra shows that reasoning about the order of messages in concurrent systems is quite difficult. Nierstrasz [11] proposes "regular types" and "request substitutability" as foundations of subtyping. His very general results are not concrete enough to develop a static type system from them. A similar definition of subtyping was given by Bowman et al. [2]. The proposal of Nielson and Nielson [10] can deal with constraints on the ordering of messages. Their type system cannot ensure that all sent messages are understood. But, subtyping is supported so that instances of subtypes preserve the properties expressed in supertypes.

The work of Liskov and Wing on behavioral subtyping [6] shows the importance of "constraints" in subtyping: Assertions on methods are not sufficient to specify an object's behavior at the presence of subtyping and aliases.

Process types as presented in [14, 15] improve previous work on process types represented as expressions in a process calculus [13]. The present work increases the expressiveness of process types by allowing the acceptability of messages to depend on exact numbers of available tokens. Other than in previous work, the type equivalence relation and subtyping relation are shown to be equivalent to an extensibility-equivalence relation and an extensibility-substitutability relation, respectively. The object calculus and type checking rules presented in [14, 15] can be used together with non-regular process types.

The proposal of Najm and Nimour [9] has a similar purpose as process types. However, in their approach at each time only one client is allowed to interact with a server through an interface specifying acceptable message changes. Process types do not have this restriction because of type splitting.

6 Conclusions

The expressiveness of process types need not be restricted to regular sets of acceptable message sequences. Type equivalence and subtyping are decidable, sound and complete for non-regular process types, provided that these relations conform to an extensibility criterion.

Acknowledgments

Many thanks to Christof Peter for his useful comments. This work was supported by the FWF (Fonds zur Förderung wissenschaftlicher Forschung) under project number P12703-INF (Static Process Types for Active Objects).

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. In *Conference Record of the 18th Symposium on Principles of Programming Languages*, pages 104–118. ACM, 1991.

[2] H. Bowman, C. Briscoe-Smith, J. Derrick, and B. Strulo. On behavioural subtyping in LOTOS. In *Proceedings FMOODS '97*, Canterbury, United Kingdom, July 1997.

[3] J.-L. Colaco, M. Pantel, and P. Salle. A set-constraint-based analysis of actors. In *Proceedings FMOODS '97*, Canterbury, United Kingdom, July 1997. Chapman & Hall.

[4] S. J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Conference Record of the 20th Symposium on Principles of Programming Languages*, Jan. 1993.

[5] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, 1969.

[6] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[7] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, Dept. of Comp. Sci., Edinburgh University, 1991.

[8] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.

[9] E. Najm and A. Nimour. A calculus of object bindings. In *Proceedings FMOODS '97*, Canterbury, United Kingdom, July 1997.

[10] F. Nielson and H. R. Nielson. From CML to process algebras. In *Proceedings CONCUR'93*, number 715 in Lecture Notes in Computer Science, pages 493–508. Springer-Verlag, 1993.

[11] O. Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28(10):1–15, Oct. 1993. Proceedings OOPSLA'93.

[12] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings LICS'93*, 1993.

[13] F. Puntigam. Types for active objects based on trace semantics. In E. N. et al., editor, *Proceedings FMOODS '96*, Paris, France, Mar. 1996. IFIP WG 6.1, Chapman & Hall.

[14] F. Puntigam. Coordination requirements expressed in types for active objects. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, number 1241 in Lecture Notes in Computer Science, Jyväskylä, Finland, June 1997. Springer-Verlag.

[15] F. Puntigam. Dynamic type information in process types. In D. Pritchard and J. Reeve, editors, *Proceedings EuroPar '98*, number 1470 in Lecture Notes in Computer Science, Southampton, England, Sept. 1998. Springer-Verlag.

[16] F. Puntigam. Non-regular process types. Technical report, Institut für Computersprachen, Technische Universität Wien, Vienna, Austria, 1999.

[17] A. Ravara and V. T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Proceedings Euro-Par '97*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[18] V. T. Vasconcelos. Typed concurrent objects. In *Proceedings ECOOP'94*, number 821 in Lecture Notes in Computer Science, pages 100–117. Springer-Verlag, 1994.

[19] V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic pi-calculus. In *Proceedings CONCUR'93*, July 1993.