A Generic Type System for Higher-Order Ψ-calculi

Alex Rønning Bendixen
Department of Computer Science
Aalborg University, Denmark

Bjarke Bredow Bojesen
Department of Computer Science
Aalborg University, Denmark

Hans Hüttel
Department of Computer Science
University of Copenhagen, Denmark
Department of Computer Science
Aalborg University, Denmark
hans.huttel@di.ku.dk

Stian Lybech*
Department of Computer Science
Reykjavík University, Iceland
stian21@ru.is

The Higher-Order Ψ-calculus framework (HOΨ) is a generalisation of many first- and higher-order extensions of the π-calculus. It was proposed by Parrow et al. who showed that higher-order calculi such as HOπ and CHOCS can be expressed as HOΨ-calculi. In this paper we present a generic type system for HOΨ-calculi which extends previous work by Hüttel on a generic type system for first-order Ψ-calculi. Our generic type system satisfies the usual property of subject reduction and can be instantiated to yield type systems for variants of HOπ, including the type system for termination due to Demangeon et al.. Moreover, we derive a type system for the ρ-calculus, a reflective higher-order calculus proposed by Meredith and Radestock. This establishes that our generic type system is richer than its predecessor, as the ρ-calculus cannot be encoded in the π-calculus in a way that satisfies standard criteria of encodability.

1 Introduction

Process calculi are formalisms for modelling and reasoning about concurrent and distributed computations; a prominent example being the π-calculus of Milner et al. [19, 25], which models computation as communication between processes, by passing messages on named channels. Since its inception, a multitude of variants of the π-calculus have appeared; e.g. Dπ [11], the calculus of explicit fusions [9], the spi-calculus with correspondence assertions [1] and the $^{e}\pi$-calculus [6]. These calculi are all *first-order*, in the sense that only atomic channel names can be passed around, not processes themselves. Bengtson et al. [3, 4] created Ψ-*calculi* as a generalisation of these first-order variants and extensions, allowing a range of calculi, including all of the aforementioned, to be expressed as *instances* of the Ψ-calculus framework through appropriate settings of a small number of parameters. However, there also exist *higher-order* variants of the π-calculus, such as the Higher-Order π-calculus, HOπ, [24, 23], that also allow *processes* to be sent across channels. Parrow et al. [21] have extended the Ψ-calculus framework with a construct for higher-order communication, creating the *Higher-Order* Ψ-calculus, HOΨ. Calculi such as HOπ and CHOCS [26] can now be represented as HOΨ-instances, as well as every calculus that the 'first-order' Ψ-calculus framework can represent.

One of the techniques for reasoning about processes is that of *type systems*. The first type system for a process calculus is due to Milner [19] and deals with the notion of correct usage of channels in the π-calculus: In a well-typed process only names of the correct type can be communicated. Pierce and Sangiorgi [22] later described a type system that uses subtyping and capability tags to control the use

*The work by this author was partly supported by the Icelandic Research Fund Grant No. 218202-05(1-3).

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on
Expressiveness in Concurrency and Structural Operational Semantics 2022
(EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 43–59, doi:10.4204/EPTCS.368.3

of names as input or output channels; and also many of the aforementioned first-order extensions of the π-calculus have been given type systems to capture such properties as secrecy, authenticity and access control.

In [12], Hüttel noted that these type systems, despite arising in different settings, share certain characteristics: The type judgments for processes P are all of the form $\Gamma \vdash P$ where Γ is a type environment recording the types of the free names in P, so processes are only classified as being either well-typed or not. On the other hand, terms M are given a type T, so type judgements for terms are of the form $\Gamma \vdash M : T$. Based on these shared characteristics, Hüttel then created a generic type system for the first-order Ψ-calculus framework, that generalises several of the type systems for the π-calculus and its variants. This generic type system can similarly be instantiated through parameter settings to yield both well-known and new type systems for the calculi that are representable as first-order Ψ-calculi. An important advantage of this approach is that a general result of type system soundness can be formulated, which is then inherited by all instances of the type system.

There has been some other works on generic type systems, notably those of König [16], Caires [5] and Igarashi and Kobayashi [15]. However, these are formulated for variants of the first-order π-calculus, which thus limits their applicability to languages that can be represented in the first-order paradigm. Stated otherwise, they exclude languages such as the aforementioned ${}^{e}\pi$-calculus, which cannot be encoded into the first-order π-calculus, as shown in [6], but which nevertheless can be given a type system using the generic approach of Hüttel, indicating that the latter is a more general framework for first-order calculi.

However, the generic type system of Hüttel can only type *first-order* calculi; it cannot be instantiated to yield type system for *higher-order* calculi, such as HOπ or CHOCS. Both of these higher-order calculi can be encoded into the first-order π-calculus, as shown by Sangiorgi in [24], and may therefore also be represented in just the first-order Ψ-calculus. Not surprisingly, there is therefore little work on type systems for higher-order calculi, since these encodings allow us to disregard the higher-order behaviour and instead just type the first-order translations. One exception is the type system for termination in variants of HOπ, due to Demangeon et al. [7]. As these authors argue, it may not always be desirable (or even possible) to type a higher-order language through a first-order representation, if the language contains features that are difficult (or impossible) to encode. For example, higher-order behaviour may alternatively be viewed as a special case of *reflection*; i.e. the ability of a program to turn code into data, modify or compute with it, and reinstantiate it as running code; and process mobility here appears as a special case where data (code/processes) are transmitted without modification.

This reflective capability is inherent in the Reflective Higher-Order (RHO or ρ) calculus of Meredith and Radestock [18], and this calculus cannot be uniformly encoded in the π-calculus, as shown in [17]. This calculus therefore gives us an example of a language that cannot easily be represented in the first-order paradigm, thus making it difficult (or impossible) to adapt any of the existing first-order type systems to this language. Yet it *can* be instantiated as a HOΨ-calculus, as we shall show in the following.

The goal of the present paper is therefore to extend the aforementioned generic type system by Hüttel, to create a generic type system for the HOΨ-calculus framework that will allow us to capture typability in the higher-order paradigm. It allows us to identify what should be required of type systems for higher-order process calculi that are instances of the HOΨ-calculus, and these requirements here take the form of a number of assumptions that must hold for each instance. Like its predecessor, our generic type system also satisfies a general subject reduction property that is inherited by all instances. We use this to formulate simple type systems for HOπ, and we show that the type system for termination by Demangeon et al. also can be captured as an instance of our type system. Lastly, we show that our generic type system can be instantiated to yield a type system for the ρ-calculus, which establishes that our type system is richer than the first-order type systems. To our knowledge, no type system has hitherto been

published for this calculus, so we regard this instance as a further contribution of the present paper. A technical report with full proofs of most results is available in [2].

2 The higher-order Ψ-calculus

The Higher-Order Ψ-calculus extends the original Ψ-calculus [3] with primitives for higher-order communication, i.e. process mobility. In this section we first review the syntax of HOΨ as given in [21], and then proceed to give a reduction semantics for the calculus.

2.1 Syntax

The Higher-Order Ψ-calculus is a general framework, which is intended to allow many different calculi to be obtained as instances, by setting a small number of parameters which takes the form of definitions of three (not necessarily disjoint) sets of *terms, conditions* and *assertions*. To allow the framework to be as general and flexible as possible, the authors of [3, 21] identify only a few restrictions that must be imposed on these sets: they must be *nominal datatypes*. Informally, a *nominal set*, in the sense of Gabbay and Pitts [8], is a set whose members can be affected by names being bound or swapped. If a, b are names and X is an element of a nominal set, then the *transposition* of a and b on X, written $(a,b)\cdot X$, swaps all occurrences of a for b in X and vice versa. A function on a nominal set is *equivariant*, if it is unaffected by name swapping; and a *nominal datatype* is a nominal set together with a set of equivariant functions on it. This requirement is very mild and allows e.g. non-well-founded sets to be used in an instantiation. The utility of this shall become apparent later, when we create a HOΨ-calculus instance where the set of processes (which itself contains terms) is included in the set of terms.

Another important notion is that of support: if X is an element of a nominal set, the *support* of X, written $\mathrm{n}(X)$, is the set of names that occur in X. Conversely, a name a is *fresh for* X, written $a\#X$, if $a \notin \mathrm{n}(X)$; and we extend this to sets of names A such that $A\#X$ if it is the case that $\forall a \in A.a \notin \mathrm{n}(X)$. This is pointwise extended to lists of elements $X_1,\ldots,X_n$, so we write $A\#X_1,\ldots,X_n$ for $A\#X_1 \wedge \ldots \wedge A\#X_n$.

As mentioned above, any Ψ-calculus instance requires a specification of three nominal datatypes: the terms, conditions and assertions. The datatype of *terms*, ranged over by $M,N \in \mathbb{T}$, contains the terms that can be communicated and used as channels. These could be e.g. single names, as in the monadic π-calculus, vectors of names as in ${}^e\pi$ and the polyadic π-calculus; or elements of a composite datatype (e.g. the integers). The datatype of *conditions*, ranged over by $\varphi \in \mathbb{C}$ contains the conditions that can be used in conditional process expressions. Finally, and importantly, we have the nominal datatype of *assertions*, ranged over by $\Psi \in \mathbb{A}$. Each of the datatypes $\mathbb{T}$, $\mathbb{C}$ and $\mathbb{A}$ must include an equivariant *substitution function*, written $(\cdot)[\widetilde{a} := \widetilde{M}]$, substituting tuples of terms $\widetilde{M}$ for tuples of names $\widetilde{a}$ of equal arity. It must be defined such that it satisfies the following *substitution laws:*

1. If $\widetilde{a} \subseteq \mathrm{n}(X)$ and $b \in \mathrm{n}(\widetilde{Y})$ then $b \in \mathrm{n}\Big(X[\widetilde{a} := \widetilde{Y}]\Big)$
2. If $\widetilde{u}\#X, \widetilde{v}$ then $X[\widetilde{v} := \widetilde{Y}] = \big((\widetilde{u},\widetilde{v})\cdot X\big)[\widetilde{u} := \widetilde{Y}]$

The requirements are quite general and should be satisfied by any ordinary definition of substitution: The first law states that names cannot be lost in substitution, i.e. the names present in $\widetilde{Y}$ must also be present when the substitution has been performed; whilst the second law states that substitution cannot be affected by transposition.

Since the calculus allows arbitrary terms to be used as channels, any Ψ-calculus instance requires a definition of two equivariant operators, *channel equivalence* $\leftrightarrow$ and *assertion composition* $\otimes$, a *unit*

element 1 of assertions, and an *entailment relation* $\Vdash$, defined on the respective nominal datatypes and with the following signatures:

$$\begin{array}{ll} \leftrightarrows : \mathbb{T} \times \mathbb{T} \to \mathbb{C} \text{ channel equivalence} & 1 \in \mathbb{A} \text{ assertion unit} \\ \otimes : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \text{ assertion composition} & \Vdash \subseteq \mathbb{A} \times \mathbb{C} \text{ entailment relation} \end{array}$$

We write the entailment relation as $\Psi \Vdash \varphi$ instead of $(\Psi, \varphi) \in \Vdash$ to denote that the condition φ holds, given the assertions Ψ. Note that comparison by channel equivalence $M_1 \leftrightarrows M_2$ is itself a condition, which may or may not be entailed by some assertions Ψ, according to the definition of the entailment relation.

The set of HOΨ-calculus *processes* $\mathscr{P}_\Psi$ is generated by the formation rules:

$$\begin{array}{rl} P \in \mathscr{P}_\Psi ::= & \mathbf{0} \mid P_1 \mid P_2 \mid \overline{M}N.P \mid \underline{M}(\lambda \widetilde{x} : \widetilde{T})N.P \\ & \mid \mathbf{run}\, M \mid \mathbf{case}\, \widetilde{\varphi} : \widetilde{P} \mid (\nu x : T)\, P \mid\ !P \mid (\!|\Psi|\!) \end{array}$$

where $\widetilde{\varphi} : \widetilde{P} \triangleq \varphi_1 : P_1 \ [] \ \ldots \ [] \ \varphi_n : P_n$.

Most of these constructs are similar to those of the π-calculus; the input and output prefixes generalise those of the π-calculus, since here both subject and object are *terms* rather than just names. Thus $\overline{M}N.P$ outputs the term N on M and continues as P, whilst $\underline{M}(\lambda \widetilde{x} : \widetilde{T})N.P$ receives a term (e.g. K) on M that must match the pattern N. Here, $\widetilde{x}$ is a list of pattern variables, binding into N and P, that is used to extract subterms from K that will then be substituted for the occurrences of $\widetilde{x}$ within the continuation P. Unlike the presentation in [21], we here use a typed version of the language: thus the types of the pattern variables are found in the list $\widetilde{T}$ where $|\widetilde{x}| = |\widetilde{T}|$, and likewise, in the restriction $(\nu x : T)\, P$, we annotate the name x bound in P with its type T.

The selection construct $\mathbf{case}\, \widetilde{\varphi} : \widetilde{P}$ is a shorthand for a list of cases and is to be understood as saying: If condition φ_i is entailed by the assertions Ψ, we continue as P_i. If more than one condition is entailed, the process is chosen non-deterministically. This construct thus generalises the choice and matching operators of the π-calculus.

Higher-order communication is handled by representing processes as terms, thus allowing them to be communicated. We assume the existence of assertions of the form $M \Leftarrow P$. By writing such an assertion, M becomes a *handle* of the process P, and we can then send P by sending its handle. Thence M may be used to activate the process P, and for this we use the only construct that is new to the higher-order setting, the invocation construct $\mathbf{run}\, M$. Note that the set of processes may itself be included in the set of terms, thus allowing assertions of the form $P \Leftarrow P$ whereby a process becomes a handle for itself.

Lastly, an assertion $(\!|\Psi|\!)$ is said to be *guarded*, if it occurs as a subterm of an input or output, and *unguarded* otherwise. The authors in [21] impose the restriction that no assertion may occur unguarded in the processes in a conditional expression $\mathbf{case}\, \widetilde{\varphi} : \widetilde{P}$, nor in a replicated process $!P$, nor in processes spawned by a $\mathbf{run}\, M$ operator. We say that processes conforming to this criterion are *well-formed*, and we shall only consider well-formed processes in the following.

2.2 Reduction semantics

Unlike previous presentations such as [4, 21] we here use reduction semantics, as this will simplify our account of the generic type system. As in other reduction semantics for process calculi, we introduce a notion of structural congruence, $\equiv_S$, as the least congruence on process terms containing α-equivalence, the commutative monoidal rules for parallel composition, and the rule for scope extrusion:

$$[\text{S-SCOPE}]\ (\nu x : T)\, P \mid Q \equiv_S (\nu x : T)\, (P \mid Q) \quad \text{if } x \# Q$$

$$[\text{E-RES}]\frac{\Psi \rhd P \underline{\ggg} P'}{\Psi \rhd (\nu x:T)P \underline{\ggg} (\nu x:T)P'}\ (x\#\Psi) \qquad [\text{E-CASE}]\frac{\Psi \Vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \underline{\ggg} P_i}$$

$$[\text{E-STRUCT}]\frac{P \equiv_S P'}{\Psi \rhd P \underline{\ggg} P'} \qquad [\text{E-RUN}]\frac{\Psi \Vdash M \Leftarrow P}{\Psi \rhd \mathbf{run}\ M \underline{\ggg} P}$$

$$[\text{E-PAR}]\frac{\Psi \otimes \mathscr{F}_\Psi(Q) \rhd P \underline{\ggg} P'}{\Psi \rhd P \mid Q \underline{\ggg} P' \mid Q}\ (\mathscr{F}_\nu(Q)\#\Psi, \mathscr{F}_\nu(P), P) \qquad [\text{E-REP}]\frac{}{\Psi \rhd\ !P \underline{\ggg} P \mid\ !P}$$

$$[\text{R-COM}]\frac{\Psi \Vdash M \leftrightarrows K}{\Psi \rhd \overline{M}N[\widetilde{x} := \widetilde{L}].P \mid \underline{K}(\lambda \widetilde{x}:\widetilde{T})N.Q \to P \mid Q[\widetilde{x} := \widetilde{L}]}$$

$$[\text{R-EVAL}]\frac{\Psi \rhd P \underline{\ggg} Q \quad \Psi \rhd Q \to P'}{\Psi \rhd P \to P'} \qquad [\text{R-RES}]\frac{\Psi \rhd P \to P'}{\Psi \rhd (\nu x:T)P \to (\nu x:T)P'}\ (x\#\Psi)$$

$$[\text{R-PAR}]\frac{\Psi \otimes \mathscr{F}_\Psi(Q) \rhd P \to P'}{\Psi \rhd P \mid Q \to P' \mid Q}\ (\mathscr{F}_\nu(Q)\#\Psi, \mathscr{F}_\nu(P), P)$$

Figure 1: Reduction semantics for the HOΨ-calculus

We introduce a parametrised, asymmetric *evaluation relation* $\cdot \rhd \cdot \underline{\ggg} \cdot$ to properly handle case expressions and unfolding of **run** M terms, both of which may depend on the assertions currently in effect. It replaces the usual structural congruence rule in the reduction semantics to ensure that neither of these operations may be reversed by a reverse reading of the rules, whilst including $\equiv_S$ for the other kinds of process rewrites where symmetry is unproblematic. The *reduction relation* $\cdot \rhd \cdot \to \cdot$, including the evaluation relation, is then given by the rules in figure 1, and reductions are thus on the form $\Psi \rhd P \to P'$, i.e. relative to a global Ψ containing the assertions currently in effect.

New assertions $(\!|\Psi|\!)$ may also appear in the syntax and therefore become enabled during the evolution of the program. These are collected by the *frame function* $\mathscr{F}_\Psi(P)$ in the [R-PAR] and [E-PAR] rules; and likewise are any new names $(\nu x:T)$ collected by $\mathscr{F}_\nu(P)$, used in the side conditions to ensure freshness of x w.r.t. Ψ and the process in parallel composition. The relevant clauses for $\mathscr{F}_\Psi(P)$ and $\mathscr{F}_\nu(P)$ are:

$$\begin{aligned} \mathscr{F}_\Psi(P \mid Q) &\triangleq \mathscr{F}_\Psi(P) \otimes \mathscr{F}_\Psi(Q) & \qquad \mathscr{F}_\nu(P \mid Q) &\triangleq \mathscr{F}_\nu(P) \cup \mathscr{F}_\nu(Q) \\ \mathscr{F}_\Psi((\nu x:T)P) &\triangleq \mathscr{F}_\Psi(P) & \qquad \mathscr{F}_\nu((\nu x:T)P) &\triangleq \{x\} \cup \mathscr{F}_\nu(P) \\ \mathscr{F}_\Psi((\!|\Psi|\!)) &\triangleq \Psi \end{aligned}$$

and with all remaining clauses of the forms $\mathscr{F}_\Psi(P) \triangleq 1$ and $\mathscr{F}_\nu(P) \triangleq \emptyset$ respectively.

3 The generic type system

The goal of our generic type system is to be able to instantiate it such that we obtain a sound type system for a given HOΨ-calculus instance. As in other type systems, we need to describe when processes are well-typed, but since we in the HOΨ-calculus also have terms, conditions and assertions, we shall therefore also need a way to decide when *they* are well-typed. However, since these nominal datatypes are parameters to the HOΨ-calculus, we cannot specify a set of type rules for them, as we can with processes.

Instead, such rules must likewise be provided as parameters to create an instance of the generic type system, and these rules must then satisfy a number of requirements, here denoted *instance assumptions*, which we shall need in the proof for subject reduction. We describe them in detail below, in section 3.3.

3.1 Types and type judgements

Types can contain names, and we assume that the set of types ***Types*** is a nominal datatype ranged over by T; however, we do not allow substitution of terms for names *inside* types.[1] Furthermore, we need the concept of a type environment Γ to record the types of free names; thus Γ is a partial function with finite support $\Gamma : \mathcal{N} \rightharpoonup \textbf{\textit{Types}}$. We can think of Γ as a set of tuples $\Gamma \subseteq \mathcal{N} \times \textbf{\textit{Types}}$ where $(x,T) \in \Gamma$ if $\Gamma(x) = T$, and we write $\Gamma, x : T$ to denote the type environment Γ extended by the name x with type T.

As usual, our type judgments will be relative to a type environment Γ. However, due to the presence of assertions which may affect the well-typedness of a process, term, condition, or indeed an assertion, our type judgments must also be relative to a global assertion Ψ. As it may be composed with assertions appearing in a process, we shall therefore also need the notion of a specialisation preorder on assertions. We say that $\Psi_1 \leq \Psi_2$ if there exists a Ψ such that $\Psi_2 = \Psi_1 \otimes \Psi$, and $\mathrm{n}(\Psi_1) \subseteq \mathrm{n}(\Psi_2)$.

Given the above, type judgements for processes will be of the form $\Gamma, \Psi \vdash P$. As previously mentioned, the type rules for terms, assertions and conditions will depend on how these parameters are defined for a specific instance of the HOΨ-calculus, and they must therefore be provided as part of the instantiation of the generic type system. However, like type judgments for processes, they must also be relative to a type environment Γ and a global Ψ, so we require that they be of the form $\Gamma, \Psi \vdash \mathscr{J}$, where $\mathscr{J}$ is defined by the formation rules:

$$\mathscr{J} \triangleq M : T \;\Big|\; \varphi \;\Big|\; \Psi$$

3.2 Channel compatibility

When we type an input or output prefix term, the type of the subject M and the type of the object (the term transmitted on channel M) must be compatible w.r.t. a *compatibility predicate* $\hookleftarrow\!\!\rho$ that describes which types of values can be carried by channels of a given types. Thus, $T_1 \hookleftarrow\!\!\rho\, T_2$ denotes that channels of type T_1 can carry terms of type T_2, and we require that the set of types be defined such that this holds. Furthermore, we distinguish between output compatibility $\hookleftarrow\!\!\rho^+$, and input compatibility $\hookleftarrow\!\!\rho^-$, and we write $T_1 \hookleftarrow\!\!\rho\, T_2$ if both $T_1 \hookleftarrow\!\!\rho^+ T_2$ and $T_1 \hookleftarrow\!\!\rho^- T_2$.

As an example, consider the channel types in the sorting system by Milner [19]. Here, a name has type $\mathrm{ch}(T)$, if it is a channel that can be used to transmit names of type T, so in that case we would therefore require that $\mathrm{ch}(T) \hookleftarrow\!\!\rho\, T$.

In our definition of compatibility, we assume given a subtype ordering $\leq$ on types. If $T_1 \leq T_2$, then a term of T_1 can be used wherever a term of type T_2 is needed. Thus we require the usual subsumption rule for types, namely that a term of a given type T_1 can also be typed with a supertype T_2:

$$[\textsc{subsume}]\frac{\Gamma, \Psi \vdash M : T_1 \quad T_1 \leq T_2}{\Gamma, \Psi \vdash M : T_2}$$

The compatibility predicate for a type T must further satisfy the following requirements w.r.t. the subtyping relation:

[1] As we shall see in our examples of instantiations of the type system, a type T may itself contain a type environment Γ, which thus may contain names with type annotations. By this requirement, we disallow that such names may be substituted for terms.

$$
\begin{aligned}
[\text{T-ENV-WEAK}]\ & \Gamma,\Psi \vdash \mathscr{J} \implies \Gamma,x:T,\Psi \vdash \mathscr{J} \\
[\text{T-ENV-STRENGTH}]\ & \Gamma,x:T,\Psi \vdash \mathscr{J} \wedge x \notin \mathrm{n}(\mathscr{J}) \implies \Gamma,\Psi \vdash \mathscr{J} \\
[\text{T-COMP-TERM}]\ & \Gamma,\Psi \vdash M[\widetilde{x}:=\widetilde{L}] : F(\widetilde{T}) \implies \Gamma,\Psi \vdash \widetilde{L}:\widetilde{T} \\
[\text{T-ASS-WEAK}]\ & \Gamma,\Psi \vdash \mathscr{J} \wedge \Psi \leq \Psi' \wedge \mathrm{n}(\Psi') \subseteq \mathrm{dom}(\Gamma) \implies \Gamma,\Psi' \vdash \mathscr{J} \\
[\text{T-WEAK-CHANEQ}]\ & \Psi \Vdash M_1 \leftrightarrows M_2 \implies \Psi \otimes \Psi' \Vdash M_1 \leftrightarrows M_2 \\
[\text{T-SUBS}]\ & \Gamma,\Psi \vdash \widetilde{L}:\widetilde{T} \wedge \Gamma,\widetilde{x}:\widetilde{T},\Psi \vdash \mathscr{J} \implies \Gamma,\Psi \vdash \mathscr{J}[\widetilde{x}:=\widetilde{L}] \\
[\text{T-EQUAL}]\ & \Gamma,\Psi \vdash M:T \wedge \Psi \Vdash M \leftrightarrows N \implies \Gamma,\Psi \vdash N:T \\
[\text{T-ENV-CLAUS}]\ & \Gamma,\Psi \vdash M:T \wedge T \curvearrowleft \Gamma' \implies \mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma') \\
[\text{T-WEAK-ASS-CLAUS}]\ & \Psi \Vdash M \Leftarrow P \wedge \Gamma,\Psi \vdash M \Leftarrow P \wedge \Psi \leq \Psi' \wedge \mathrm{n}(\Psi) \subseteq \Gamma \implies \Psi' \Vdash M \Leftarrow P \\
[\text{T-SUBS-RUN}]\ & \Gamma,\Psi \vdash M:T \wedge T \curvearrowleft \Gamma' \wedge \Psi \Vdash M[\widetilde{x}:=\widetilde{L}] \Leftarrow P \implies \Gamma',\Psi \vdash P
\end{aligned}
$$

Figure 2: Instance assumptions for the generic type system.

1. If a channel type can carry two distinct types, then the types have to be related by the subtype ordering. That is, if $d \in \{+,-\}$, $T \leftrightarrow\!\!\!\rho^d\ T_1$ and $T \leftrightarrow\!\!\!\rho^d\ T_2$ with $T_1 \neq T_2$, then $T_1 \leq T_2$ or $T_2 \leq T_1$.
2. Output compatibility is *contravariant*. That is, if $T \leftrightarrow\!\!\!\rho^+\ T_2$ and $T_1 \leq T_2$, then also $T \leftrightarrow\!\!\!\rho^+\ T_1$. This requirement mirrors that of [22]. If $T_1 \leq T_2$, then a term of type T_1 can be used where ever a term of type T_2 is needed, and a channel that outputs terms of the more general type T_2 can therefore be used, where ever a channel of the specialized type T_1 is required.
3. Input compatibility is *covariant*. That is, if $T \leftrightarrow\!\!\!\rho^-\ T_1$ and $T_1 \leq T_2$, then also $T \leftrightarrow\!\!\!\rho^-\ T_2$. This requirement, too, mirrors that of [22]. Here, if $T_1 \leq T_2$, a channel that accepts terms of type T_1 can also be used to accept terms of type T_2.

3.3 Instance assumptions

In order to ensure soundness, we introduce a collection of assumptions, given in Figure 2, that must hold for an instance of the generic type system to be valid. They pertain to the type judgments $\Gamma,\Psi \vdash \mathscr{J}$ for terms, conditions and assertions, which, as previously mentioned, we cannot specify in advance, but on which we must nevertheless impose certain restrictions to allow us to prove subject reduction for the generic type system. Specifically, the assumptions will guarantee that the properties of weakening and strengthening and the substitution lemma will hold for any instance that satisfies these assumptions.

Firstly, we require every instance of our generic type system to satisfy certain natural requirements about the use of type environments Γ; these are similar to those of Hüttel in [12]. The assumptions [T-ENV-WEAK], [T-ENV-STRENGTH], [T-COMP-TERM] and [T-ASS-WEAK] are the usual requirements of weakening and strengthening; these must hold for type environments as well as for assertions. [T-WEAK-CHANEQ] tells us that channel equivalence is closed under weakening of assertions. The assumptions [T-SUBS] and [T-EQUAL] tell us that typability must be invariant under substitution and channel equivalence.

Other assumptions are particular to the higher-order setting and thus new. Here, one particularly important question is *which* type environment Γ a process P should be typed in relation to, if P is transmitted using the higher-order process mobility construct, with M as a handle for P. To solve this, we write $T \curvearrowleft \Gamma$ to express that if M is a handle for some process P and has type T, then we can *extract*

$$[\text{T-IN}]\frac{T \leftrightharpoons T' \quad \Gamma,\Psi\vdash M:T \quad \Gamma,\widetilde{x}:\widetilde{T},\Psi\vdash N:T' \quad \Gamma,\widetilde{x}:\widetilde{T},\Psi\vdash P}{\Gamma,\Psi\vdash \underline{M}(\lambda\widetilde{x}:\widetilde{T})N.P} \qquad [\text{T-RUN}]\frac{T \curvearrowleft \Gamma' \quad \Gamma,\Psi\vdash M:T \quad \Psi \Vdash M \Leftarrow P \quad \Gamma',\Psi\vdash P}{\Gamma,\Psi\vdash \mathbf{run}\ M} \qquad [\text{T-OUT}]\frac{T \leftrightharpoons T' \quad \Gamma,\Psi\vdash M:T \quad \Gamma,\Psi\vdash N:T' \quad \Gamma,\Psi\vdash P}{\Gamma,\Psi\vdash \overline{M}N.P}$$

$$[\text{T-PAR}]\frac{\Gamma,\mathscr{F}_{\mathcal{V}}(Q),\Psi\otimes\mathscr{F}_{\Psi}(Q)\vdash P \quad \Gamma,\mathscr{F}_{\mathcal{V}}(P),\Psi\otimes\mathscr{F}_{\Psi}(P)\vdash Q}{\Gamma,\Psi\vdash P \mid Q}\ \begin{pmatrix}\mathscr{F}_{\mathcal{V}}(P)\,\#\Psi, \mathscr{F}_{\mathcal{V}}(Q), Q\\ \mathscr{F}_{\mathcal{V}}(Q)\,\#\Psi, \mathscr{F}_{\mathcal{V}}(P), P\end{pmatrix}$$

$$[\text{T-NEW}]\frac{\Gamma,x:T,\Psi\vdash P}{\Gamma,\Psi\vdash(\nu x:T)P}\ (x\#\Psi) \qquad [\text{T-NIL}]\frac{}{\Gamma,\Psi\vdash\mathbf{0}} \qquad [\text{T-REPL}]\frac{\Gamma,\Psi\vdash P}{\Gamma,\Psi\vdash\,!P}$$

$$[\text{T-CASE}]\frac{\Gamma,\Psi\vdash\varphi_i \quad \Gamma,\Psi\vdash P_i}{\Gamma,\Psi\vdash\mathbf{case}\ \widetilde{\varphi}:\widetilde{P}} \qquad [\text{T-ASSERT}]\frac{\Gamma,\Psi\vdash\Psi'}{\Gamma,\Psi\vdash(\!|\Psi'|\!)}$$

Figure 3: Type judgements for processes

the type environment Γ for typing P from the type T of the handle M. This is thus another requirement we impose on how the set of types must be defined. As a simple example, suppose that every type of a term would consist of a channel component and a run type component (T,Γ); then we could define the $\curvearrowleft$ relation to be $(T,\Gamma)\curvearrowleft\Gamma$.

The new assumptions are as follows:

- The assumption [T-ENV-CLAUS] tells us that that the type environment extracted from the type of a handle M must mention at least the free names of M.
- The assumption [T-WEAK-ASS-CLAUS] is necessary to prove weakening of assertion environments; i.e. by allowing unused assertions to be added. It states that if M is a handle for P, then M must still remain a handle for the same process P if the assertion environment is weakened.
- The assumption [T-SUBS-RUN] is needed to ensure that typability is preserved by substitution also in the higher-order case. It states that if a term M becomes a handle for a new process P after a substitution, then the new process must still be well-typed in the environment we obtain from M's type T.

3.4 Type rules for processes

Unlike the aforementioned type rules for terms, conditions and assertions, the type rules for processes are common to every instance. As in [12], we only consider type judgements that are *well-formed*; that is, if $\mathrm{n}(\Psi)\cup\mathrm{n}(P)\subseteq\mathrm{dom}(\Gamma)$, so every name mentioned in the term or process in the judgement is bound in the type environment. The rules are given in Figure 3; they are mostly similar to those of [12], except for the rule [T-RUN] used to type the **run** M construct, which is the only construct that is new to the higher-order setting.

We shall comment on the rules in some detail: In the rule for input, [T-IN], the subject M must have type T, which must be compatible with the type T' according to the aforementioned compatibility relation $\leftrightharpoons$. The pattern N must then have this type T', given the assumptions that the variables $\widetilde{x}$ have types $\widetilde{T}$,

and lastly, the continuation P must be well-typed given these assumptions. The output rule, [T-OUT] then mirrors the input rule as usual. In both cases, the type judgment $\Gamma, \Psi \vdash M : T$ appears in the premise, and as previously mentioned, the rules for this judgment must be provided as part of the instantiation.

In the rule [T-PAR] we require that for a parallel composition $P \mid Q$ to be typable, P and Q must both be typable within type environments and assertions that add information extracted from the other component; thus we here overload the function $\mathscr{F}_V(\cdot)$ for $(\nu x : T)\, P$ to mean $\mathscr{F}_V((\nu x : T)\, P) \triangleq x : T, \mathscr{F}_V(P)$. This is a natural requirement, since P can, among other things, mention handles for processes established in Q. The side condition then asserts that all new names declared in P, using the $(\nu x : T)$ construct, must be fresh for Ψ *and* both the free and new names occurring in Q, and vice versa for Q, similar to the side conditions for the [E-PAR] and [R-PAR] rules in the semantics.

Likewise in the rule [T-NEW], we require that the new name x must be fresh for Ψ, again mirroring the side conditions in the corresponding semantic rules [E-RES] and [R-RES], and P must then be well-typed given the assumption that x has type T.

The rules for the nil process and replication, [T-NIL] and [T-REPL] are as usual, and the rule for **case** $\widetilde{\varphi} : \widetilde{P}$ is also quite straightforward. Here, we write $\Gamma, \Psi \vdash \varphi_i$ and $\Gamma, \Psi \vdash P_i$ to say that every condition φ_i in the list of conditions $\widetilde{\varphi}$, and every process P_i in the list of processes $\widetilde{P}$, must be well-typed w.r.t. the same Γ and Ψ. As in the cases for input and output above, the rules for concluding $\Gamma, \Psi \vdash \varphi_i$ must be provided as part of the instantiation; and likewise for concluding $\Gamma, \Psi \vdash \Psi'$, which appears in the premise of the [T-ASSERT] rule.

Lastly, since a key motivation for the present type system is the ability to type higher-order behaviour, we must be able to describe what can happen when a handle $M \Leftarrow P$ is released by a **run** M. This is handled by the rule [T-RUN], which states that **run** M is well-typed for Γ and Ψ if M is a handle for P in Ψ and P is well-typed in the environment Γ' extracted from M, using the aforementioned $\curvearrowleft$ relation.

4 Properties of the generic type system

Type systems normally ensure two properties of well-typed programs: a *subject reduction* property guarantees that a well-typed program remains well-typed under reduction; and a *safety* property ensures that if a program is well-typed then a certain safety predicate holds. The latter will depend on the particular instance of the type system and must therefore be shown individually, for each instance, but subject reduction can be shown for the generic type system. We establish this through a series of lemmas, beginning with the usual results of *weakening* and *strengthening* of the type environment:

Lemma 1 (Weakening and strengthening)**.**

- *If* $\Gamma, \Psi \vdash P$ *then* $\Gamma, x : T, \Psi \vdash P$
- *If* $\Gamma, x : T, \Psi \vdash P$ *and* $x \# P, \Psi$ *then* $\Gamma, \Psi \vdash P$

A similar result holds for assertions. Any process that is well-typed remains well-typed after a composition of any assertion in the assertion environment, so long as all names in the new assertion environment are in the support of the type environment:

Lemma 2 (Assertion environment weakening)**.** *If* $\Gamma, \Psi \vdash P$, $\mathrm{n}(\Psi') \subseteq \mathrm{dom}(\Gamma)$ *and* $\Psi \leq \Psi'$ *then* $\Gamma, \Psi' \vdash P$.

This lemma is necessitated by the syntax of the HOΨ-calculus itself, which allows guarded assertions in continuations to become unguarded after a reduction. It is in the proof of this result that the instance assumptions [T-ASS-WEAK], [T-ENV-CLAUS] and [T-WEAK-ASS-CLAUS] become important.

As we here use reduction semantics with an asymmetric evaluation relation to handle unfolding of **case** and **run** expressions, we shall also need two results that describe how frames can evolve during

evaluation. The former, given in Lemma 3 is used in the proof of subject reduction (Theorem 1) to find any new assertions that may have become composed onto the pre-existing assertion environment after a reduction. The latter, given in Lemma 4 states that the assertions in a process are unaltered by an evaluation: This is mainly ensured by the criterion for well-formed processes, asserting that all processes under replication or in a **case** expression, and all processes spawned by a **run** M operator, may not contain unguarded assertions. The proof then establishes that the property of being assertion-guarded is preserved by the evaluation relation $\ggg$.

Lemma 3 (Frame post reduction)**.** *If* $\Psi \rhd P \to P'$ *then* $\mathscr{F}_\Psi(P) \leq \mathscr{F}_\Psi(P')$

Lemma 4 (Frame post evaluation)**.** *If* $\Psi \rhd P \underline{\ggg} P'$ *then* $\mathscr{F}_\Psi(P) = \mathscr{F}_\Psi(P')$.

The above lemmas can now be used to prove that a well-typed process remains well-typed after an evaluation:

Lemma 5 (Subject evaluation)**.** *If* $\Gamma, \Psi \vdash P \wedge \Psi \rhd P \underline{\ggg} P'$ *then* $\Gamma, \Psi \vdash P'$.

Lastly, we need a standard result of *substitution*, which states that a well-typed process remains well-typed after a well-typed substitution. The proof of this lemma requires the instance assumptions [T-SUBS] and [T-SUBS-RUN].

Lemma 6 (Subject substitution)**.** *If* $\Gamma, \widetilde{x} : \widetilde{T}, \Psi \vdash P$ *and* $\Gamma, \Psi \vdash \widetilde{L} : \widetilde{T}$ *then* $\Gamma, \Psi \vdash P[x := \widetilde{L}]$.

This, at last, gives us our main result:

Theorem 1 (Subject reduction)**.** *If* $\Gamma, \Psi \vdash P \wedge \Psi \rhd P \to P'$ *then* $\Gamma, \Psi \vdash P'$.

Outline. Induction in the reduction rules. In many of the cases, the instance assumptions are needed. An example is that in the case of the [R-COM] rule, the substitution assumption [T-SUBS] is needed to ensure that the substitution of the received message can be well-typed and the weakening assumptions [T-ENV-WEAK] and [T-ASS-WEAK] are needed to ensure that the resulting process can be typed within the same type environment as before. □

The subject reduction theorem holds for all valid instances of the generic type system. This is all that we can guarantee in our generic setting, as a notion of *safety* will also depend on a definition of runtime error, which will be specific to each instance. Safety must therefore be proved individually for each instance.

5 Instances of the generic type system

We now show how our generic type system can be applied to provide sound type systems for higher-order process calculi. We first consider type systems for a version of the HOπ-calculus [24], and then a type system for the ρ-calculus [18] introduced by Meredith and Radestock.

5.1 The Higher-Order π-calculus

Parrow et al. [21] give several examples of HOΨ-instances with process mobility: for example, by including the set of processes $\mathscr{P}_\Psi$ in $\mathbb{T}$, a process P may appear as the object of an output. If for all $P \in \mathscr{P}_\Psi . P \Leftarrow P$ is entailed by all assertions, a language similar to Thomsen's Plain CHOCS [27] is obtained, and by further allowing both names and processes to appear as objects of an output, we get a simplified version of Sangiorgi's HOπ-calculus, similar to the one described in [20]. We set the parameters for $\mathbb{T}, \mathbb{C}$ and entailment thus:

$$\mathbb{T} \triangleq \mathscr{N} \cup \mathscr{R}_\Psi$$
$$\mathbb{C} \triangleq \{\, a \leftrightarrows b \mid a,b \in \mathscr{N} \,\} \cup \{\, P \Leftarrow Q \mid P,Q \in \mathscr{R}_\Psi \,\} \cup \{\, \top \,\}$$
$$\Vdash \triangleq \{\, (1, a \leftrightarrows a) \mid a \in \mathscr{N} \,\} \cup \{\, (1, P \Leftarrow P) \mid P \in \mathscr{R}_\Psi \,\} \cup \{\, (1, \top) \,\}$$

and (initially) with $\mathbb{A} \triangleq \{\emptyset\}$, $\otimes \triangleq \cup$ and $1 \triangleq \emptyset$. We also include the symbol $\top$ in $\mathbb{C}$ to represent a condition that is entailed by all assertions, and use that for every condition in a **case** $\widetilde{\varphi} : \widetilde{P}$ construct to obtain a representation of non-deterministic choice. This parameter setting obviously allows unwanted processes such as

$$\overline{a}P.\mathbf{0} \mid \underline{a}(\lambda x)x.\overline{x}b.\mathbf{0} \to \overline{P}b.\mathbf{0}$$

where the process P is substituted for the *subject* x in the output construct $\overline{x}b.\mathbf{0}$ after a reduction step. However, we can now use our generic type system to create an instantiation that will disallow such possibilities. We define the types of terms as:

$$T \in \textbf{\textit{Types}} ::= \mathrm{ch}(T) \;\big|\; \mathrm{drop}(\Gamma)$$

The behaviour of channels and first-order variables is captured in the same manner as the simple sorting system for the π-calculus [19]. Process terms and higher-order variables will have the type $\mathrm{drop}(\Gamma)$, where the processes are well-typed in Γ. Type errors can then be expressed as a simple error predicate, with

$$\frac{\Gamma, \Psi \vdash M : \mathrm{drop}(\Gamma')}{\Gamma, \Psi \vdash \underline{M}(\lambda x)x.Q \to \mathbf{WRONG}} \qquad \frac{\Gamma, \Psi \vdash M : \mathrm{ch}(T)}{\Gamma, \Psi \vdash \mathbf{run}\ M \to \mathbf{WRONG}}$$

as the most relevant rules. We now define the instance parameters:

$$[\text{T-CON}]\frac{}{\Gamma, \Psi \vdash \top} \qquad [\text{T-ASS}]\frac{P : \mathrm{drop}(\Gamma) \in \Psi'}{\Gamma, \Psi \vdash (\!|\Psi'|\!)} \qquad [\text{T-END}]\frac{}{\mathrm{drop}(\Gamma) \curvearrowleft \Gamma}$$

$$[\text{T-CHA}]\frac{}{\mathrm{ch}(T) \leftharpoonup T} \qquad [\text{TERM}_1]\frac{\Gamma(x) = \mathrm{ch}(T)}{\Gamma, \Psi \vdash x : \mathrm{ch}(T)} \qquad [\text{TERM}_2]\frac{P : \mathrm{drop}(\Gamma') \in \Psi \quad \Gamma', \Psi \vdash P}{\Gamma, \Psi \vdash P : \mathrm{drop}(\Gamma')}$$

Here we let the type environment in a drop type be the same type environment that is exposed to the processes, when it is defined as an object in an output prefix, i.e. if we have $\Gamma, \Psi \vdash \overline{a}P$, we want $\Gamma, \Psi \vdash P : \mathrm{drop}(\Gamma)$. In this way, when we run the process, we can recall the bound variables and their types at the time when the process was sent. To implement this, we shall use the (previously unused) assertions Ψ as type environments for processes. Thus we redefine $\mathbb{A}$ as follows:

$$\mathbb{A} \triangleq \wp\big(\{\, P : T \;\big|\; P \in \mathscr{P} \wedge T \in \textbf{\textit{Types}} \,\}\big)$$

We can now show safety for the type system instance:

Theorem 2. *If* $\Gamma, \Psi \vdash P$ *then* $P \not\to$ ***WRONG***.

The proof is by induction in the rules of $\Gamma, \Psi \vdash P$. Details are given in [2].

5.2 A type system for termination

We now turn our attention to an instance of the generic type system that captures a liveness property. Demangeon et al. [7] present a type system for checking termination in variants of the HOπ-calculus: for any well-typed process P we have that $P \to^* P' \not\to$. These authors study HOpi_2, a higher-order process calculus in which only processes can be communicated. The syntax of HOpi_2 is given by the formation rules

$$P ::= \mathbf{0} \;\mid\; a(X).P \;\mid\; \overline{a}\langle Q\rangle.P \;\mid\; P_1 \mid P_2 \;\mid\; (\nu a : T)P \;\mid\; X$$

In this type system, processes P are typed with a type n, where n is a natural number called the *level* of P. Names a have types of the form $\mathrm{ch}^k(\diamond)$, where $\diamond$ denotes the type of processes and k is a natural number, the level of a. This is interpreted as saying that a is only used to carry processes whose level n is less than k. Type judgements are of the form $\Gamma \vdash P : n$. The type rules, shown below, ensure that the level of processes that are sent on any channel a will be strictly smaller than that of a.

$$[\textsc{In}]\ \frac{\Gamma, X : (k-1) \vdash P : n \quad \Gamma(a) = \mathrm{ch}^k(\diamond)}{\Gamma \vdash a(X).P : n} \qquad [\textsc{Out}]\ \frac{\Gamma \vdash Q : m \quad \Gamma \vdash P : n \quad \Gamma(a) = \mathrm{ch}^k(\diamond) \quad m < k}{\Gamma \vdash \overline{a}\langle Q\rangle.P : \max(k,n)} \qquad [\textsc{Nil}]\ \frac{}{\Gamma \vdash \mathbf{0} : 0}$$

$$[\textsc{New}]\ \frac{\Gamma, a : \mathrm{ch}^k(\diamond) \vdash P : n}{\Gamma \vdash (\nu a : T)P : n} \qquad [\textsc{Par}]\ \frac{\Gamma \vdash P : m \quad \Gamma \vdash Q : n}{\Gamma \vdash P \mid Q : \max(m,n)} \qquad [\textsc{Var}]\ \frac{\Gamma(X) = n}{\Gamma \vdash X : n}$$

It is straightforward to represent the HOpi_2 calculus as an instance of the Higher-Order Ψ-calculus, using a variant of the parameter setting described in section 5.1. In order to represent the type system, we introduce assertions of the form

$$\Psi ::= n \;\mid\; n^- \;\mid\; n^+$$

We use assertions to indicate in which way a channel is to be used; an input use can only be typed in the presence of an assertion n^- and output use must be used with an assertion n^+. We have that $n \otimes n^- = n^- \otimes n = n$; that $n \otimes n^+ = n^+ \otimes n = n$; and that $n_1 \otimes n_2 = \max(n_1, n_2)$. We distinguish explicitly between input uses ($\mathrm{ch}^k_-(\diamond)$) and output uses ($\mathrm{ch}^k_+(\diamond)$) of channels:

$$T \in \textbf{\textit{Types}} ::= n \;\mid\; \mathrm{ch}^k_-(\diamond) \;\mid\; \mathrm{ch}^k_+(\diamond)$$

and we let $\mathrm{ch}^n(\diamond) \curvearrowleft (n-1)$ and $\mathrm{ch}^n(\diamond) \curvearrowleft k$ whenever $k < n$. Type judgements are of the form $\Gamma, m \vdash M : T$ for terms and $\Gamma, m \vdash P$ for processes. We represent the judgement $\Gamma \vdash P : n$ as $\Gamma, n \vdash P$. The type rules for channels are thus:

$$[\textsc{Ch-in}]\ \frac{\Gamma(a) = \mathrm{ch}^k_-(\diamond)}{\Gamma, n^- \vdash a : \mathrm{ch}^k_-(\diamond)} \qquad [\textsc{Ch-out}]\ \frac{\Gamma(a) = \mathrm{ch}^k_+(\diamond)}{\Gamma, n^+ \vdash a : \mathrm{ch}^k_+(\diamond)}$$

5.3 The ρ-calculus

The Reflective Higher-Order calculus of Meredith and Radestock [18] is less well-known than e.g. CHOCS and HOπ, so we recall it in some detail. Unlike other calculi, the ρ-calculus does not assume an infinite set of names: instead, names and processes are both built from the same syntax, so names are structured terms, rather than atomic entities. The syntax for both processes and names is given by the formation rules:

$$\begin{aligned} P &::= \mathbf{0} \;\mid\; P \mid P \;\mid\; x\langle\!|P|\!\rangle \;\mid\; x(y).P \;\mid\; \ulcorner x \urcorner \\ x, y &::= \ulcorner P \urcorner \end{aligned}$$

where the syntax for names is simply $\ulcorner P \urcorner$, pronounced *quote P*. Names can be passed around as in the π-calculus, as well as un-quoted (called *drop*), and thus higher-order behaviour becomes an inherent property of the calculus, rather than just an extension on top of an already computationally complete language.

The parallel, and the input construct $x(y).P$, are similar to their π-calculus counterparts. The *lift* operation, $x\langle\!|P|\!\rangle$ is an output construct that quotes the process P, thereby creating the *name* $\ulcorner P \urcorner$, and sends it out on x; thus the calculus can generate new names at runtime without the need of a ν-operator. The converse of lift is the *drop* operation, $\urcorner x \ulcorner$: it is a request to run the process within a name, by removing the quotes around it. This is not performed by a reduction, but rather by a form of substitution

$$\urcorner x \ulcorner \{\ulcorner P \urcorner / x'\} = P \qquad \text{if } x \equiv_{\mathcal{N}} x'$$

where the entire *process* $\urcorner x \ulcorner$ is replaced with the process P found within the substituted name, similar to how process variables are replaced by processes in e.g. HOπ. Notably, this means that if x is a *free* name, then $\urcorner x \ulcorner$ will be a *deadlock*, since x can never be touched by a substitution at runtime. Otherwise, substitution is the standard, capture-avoiding substitution of names for names, and note in particular that substitution does *not* recur into processes under quotes; i.e. $\ulcorner P \urcorner \{x/y\} = \ulcorner P \urcorner$ if $y \not\equiv_{\mathcal{N}} \ulcorner P \urcorner$ regardless of whether the name y exists somewhere within $\ulcorner P \urcorner$.

The reduction semantics is given by the standard rules for parallel composition and structural congruence (as in e.g. the π-calculus) plus the following rule for communication:

$$[\rho\text{-COM}]\frac{x_1 \equiv_{\mathcal{N}} x_2}{x_1(y).P \mid x_2\langle\!|Q|\!\rangle \to P\{\ulcorner Q \urcorner / y\}}$$

One subtlety of this calculus concerns the notion of structural congruence, $\equiv$. It is the usual least congruence on processes, containing α-equivalence, $\equiv_\alpha$, and the abelian monoid rules for parallel composition with $\mathbf{0}$ as the unit element. However, with *structured* terms as names, $\equiv_\alpha$ in turn requires a notion of *name equivalence*, written $\equiv_{\mathcal{N}}$, that is also used for comparing subjects in the [ρ-COM] rule above. It is defined as the smallest equivalence relation on *quoted* processes, closed forward under the rules:

$$[\rho\text{-NAMEEQ}_1]\frac{P \equiv Q}{\ulcorner P \urcorner \equiv_{\mathcal{N}} \ulcorner Q \urcorner} \qquad [\rho\text{-NAMEEQ}_2]\frac{}{\ulcorner\urcorner x \ulcorner\urcorner \equiv_{\mathcal{N}} x}$$

This yields a mutual recursion between name equivalence, structural congruence and α-equivalence, albeit one that always terminates as proved in [18], because both the sets of names and processes are well-founded; their smallest elements being $\mathbf{0}$ (the inactive process) and $\ulcorner \mathbf{0} \urcorner$ respectively.

5.3.1 Instantiation as a Ψ-calculus

The ρ-calculus is interesting in the present setting, because it cannot be encoded in the π-calculus in a way that satisfies a number of generally accepted criteria of encodability, similar to those of [10]. This has been established by one of the authors in [17].

The key reason for this impossibility lies in the ability of the ρ-calculus to generate new, *free*, and hence observable, names at runtime, whilst this is not possible in the π-calculus; and, dually, its use of name equivalence, which will equate more names than strict syntactic equality. However, the ρ-calculus *can* be represented in the HOΨ-framework as follows. We define

$$\begin{aligned}\mathbb{T} &\triangleq \mathcal{N} \cup \{\ulcorner P \urcorner \mid P \in \mathscr{P}_\Psi\} \cup \{\langle\!|\ulcorner P \urcorner|\!\rangle \mid P \in \mathscr{P}_\Psi\} \\ \mathbb{C} &\triangleq \{M \dot{\leftrightarrow} N \mid M,N \in \mathbb{T}\} \cup \{P_1 \equiv P_2 \mid P_1,P_2 \in \mathscr{P}_\Psi\} \\ &\quad \cup \{M \Leftarrow P \mid M \in \mathbb{T} \wedge P \in \mathscr{P}_\Psi\}\end{aligned}$$

and (initially) with $\mathbb{A} \triangleq \{\emptyset\}$, $\otimes \triangleq \cup$ and $1 \triangleq \emptyset$ as before. Note the two different kinds of terms: we use terms of the form $\ulcorner P \urcorner$ to represent a *statically* quoted name in the ρ-calculus, which can never be dropped and never substituted into. Conversely, we use $\langle\!|\ulcorner P \urcorner|\!\rangle$ for the equivalent of the object of a $x\langle\!|P|\!\rangle$,

which in the ρ-calculus is a *process* that therefore *can* be substituted into, and which later may be dropped. Furthermore, we shall assume that all *bound names* are implemented as distinct atomic names $x \in \mathscr{N}$; this is a trivial conversion, since their structure has no semantic meaning in the ρ-calculus. The encoding is then given by the translation:

$$\begin{array}{rlrl} [\![\mathbf{0}]\!] &= \mathbf{0} & [\![\ulcorner x \urcorner]\!] &= \mathbf{run}\ x \\ [\![P_1 \mid P_2]\!] &= [\![P_1]\!] \mid [\![P_2]\!] & [\![\ulcorner\urcorner P \urcorner\urcorner]\!] &= \mathbf{0} \\ [\![n(x).P]\!] &= \underline{[\![n]\!]}(\lambda x)\langle\!\langle x \rangle\!\rangle.[\![P]\!] & [\![\ulcorner P \urcorner]\!] &= \ulcorner \mathscr{N}[\![P]\!] \urcorner \\ [\![n\langle\!\langle P \rangle\!\rangle]\!] &= \overline{[\![n]\!]}\langle\!\langle \ulcorner [\![P]\!] \urcorner \rangle\!\rangle.\mathbf{0} & [\![x]\!] &= x \end{array}$$

where $\mathscr{N}[\![P]\!]$ is similar to $[\![P]\!]$ *except* that $\mathscr{N}[\![\ulcorner\urcorner P \urcorner\urcorner]\!] = \mathbf{run}\ \ulcorner \mathscr{N}[\![P]\!] \urcorner$.

Note the two translations of drop for processes: the process $\ulcorner\urcorner P \urcorner\urcorner$ has no reduction in the ρ-calculus and is therefore behaviourally equivalent to $\mathbf{0}$; but its counterpart $\mathbf{run}\ \ulcorner P \urcorner$ *might* have a reduction, since $\mathbf{run}\ M$ is not evaluated eagerly in the HOΨ-calculus. For the purpose of preserving operational correspondence, we therefore translate the drop of a *free name* $\ulcorner P \urcorner$ as $\mathbf{0}$, and the drop of an atomic name x as $\mathbf{run}\ x$, since atomic names are bound by construction. However, we cannot do this within *names*, since name equivalence is determined by the structure, rather than the behaviour of the process within quotes. Thus we use the second level translation $\mathscr{N}[\![P]\!]$ for statically quoted names, since these can never be dropped.

Lastly, we shall define entailment such that it contains the rule $\Psi \Vdash \ulcorner P \urcorner \Leftarrow P$, making every term $\ulcorner P \urcorner$ a handle for the process P within, to mirror the duality of names and processes in the ρ-calculus. We furthermore include the following rules for entailment of *channel equivalence* $\dot{\leftrightarrow}$, mirroring the rules [ρ-NAMEEQ$_1$] and [ρ-NAMEEQ$_2$] for concluding name equivalence:

$$[\textsc{ChaneQ}_1]\frac{}{\Psi \Vdash \ulcorner \mathbf{run}\ M \urcorner \dot{\leftrightarrow} M} \qquad [\textsc{ChaneQ}_2]\frac{\Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash \ulcorner P_1 \urcorner \dot{\leftrightarrow} \ulcorner P_2 \urcorner}$$

including the symmetric and transitive closure of $\dot{\leftrightarrow}$. We then let the entailment relation for conditions of *structural congruence* $\equiv$ be defined such that $\equiv$ contains α-equivalence; that $(\mathscr{P}_{/\equiv}, \mid, \mathbf{0})$ is an abelian monoid; and containing the four congruence rules derived from the above translation:

$$[\textsc{Par}]\frac{\Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash P_1 \mid R \equiv P_2 \mid R} \qquad [\textsc{In}]\frac{\Psi \Vdash M_1 \dot{\leftrightarrow} M_2 \qquad \Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash \underline{M_1}(\lambda x_1)\langle\!\langle x_1 \rangle\!\rangle.P_1 \equiv \underline{M_2}(\lambda x_2)\langle\!\langle x_2 \rangle\!\rangle.P_2}$$

$$[\textsc{Run}]\frac{\Psi \Vdash M_1 \dot{\leftrightarrow} M_2}{\Psi \Vdash \mathbf{run}\ M_1 \equiv \mathbf{run}\ M_2} \qquad [\textsc{Out}]\frac{\Psi \Vdash M_1 \dot{\leftrightarrow} M_2 \qquad \Psi \Vdash P_1 \equiv P_2}{\Psi \Vdash \overline{M_1}\langle\!\langle \ulcorner P_1 \urcorner \rangle\!\rangle \equiv \overline{M_2}\langle\!\langle \ulcorner P_2 \urcorner \rangle\!\rangle}$$

This translation is sound and complete w.r.t. operational correspondence up to a reasonable notion of behavioural equivalence $\simeq$:

Theorem 3 (Operational correspondence)**.** *Let $\simeq$ be a notion of behavioural equivalence for processes of the* HOΨ*-instance of the ρ-calculus, that includes at least structural congruence and the axiom* $\mathbf{run}\ \ulcorner [\![P]\!] \urcorner \simeq [\![P]\!]$. *Then* $P \to P' \iff [\![P]\!] \to\simeq [\![P']\!]$.

Outline. The proof requires a number of steps. First we show that the translation preserves name equivalence; i.e. $x_1 \equiv_{\mathscr{N}} x_2 \iff 1 \Vdash [\![x_1]\!] \dot{\leftrightarrow} [\![x_2]\!]$ by induction in the rules of name equivalence and structural congruence. Then we show that substitution can be moved out of the translation; i.e. $[\![P\sigma]\!] \simeq [\![P]\!][\![\sigma]\!]$ and $1 \Vdash [\![(n)\sigma]\!] \dot{\leftrightarrow} ([\![n]\!])[\![\sigma]\!]$, where $\sigma \triangleq \{\ulcorner Q \urcorner / x\}$ and $[\![\sigma]\!] \triangleq [[\![x]\!] := [\![\ulcorner Q \urcorner]\!]]$, by induction in the clauses of the translation function. This step relies on our assumption about $\simeq$. Lastly we can show soundness and completeness w.r.t. operational correspondence by induction in the two semantics. In both cases, the interesting clauses are the communication rules, which require the aforementioned substitution and name equivalence preservation results. Details are available in [2]. □

5.3.2 A type system for reflection

Other higher-order calculi such as CHOCS and HOπ can be encoded in the π-calculus and may thus be typable through translation, but as we noted above there cannot be such an encoding of the ρ-calculus into the π-calculus. Thus, we cannot hope to create a type system for the ρ-calculus by adapting an existing first-order type system. In fact, we are not aware of any type system for the ρ-calculus, so we shall now create one by instantiating our generic type system. We let types for names be of the form

$$T \in \textbf{\textit{Types}} ::= \langle T, \Gamma \rangle \quad | \quad \langle B, \Gamma \rangle$$

where B is a basis type, and Γ is a type environment representing the possibility of executing the process within the name. Furthermore we shall use assertions as type environments for processes as we previously did with HOπ, so we update the definition accordingly.

$$\mathbb{A} \triangleq \wp(\{ \ulcorner P \urcorner : T \mid P \in \mathscr{P}_\Psi \wedge T \in \textbf{\textit{Types}} \} \cup \{ \langle\!| \ulcorner P \urcorner |\!\rangle : T \mid P \in \mathscr{P}_\Psi \wedge T \in \textbf{\textit{Types}} \})$$

with assertion unit and composition as $1 \triangleq \emptyset$ and $\otimes \triangleq \cup$ respectively. Note that by construction $\forall x \in \mathscr{N}.x\#\ulcorner P \urcorner$, so substitution can only occur in terms of the form $\langle\!| \ulcorner P \urcorner |\!\rangle : T$. We then append an assertion to the encoding of input and output:

$$\begin{aligned} [\![\ulcorner R \urcorner \langle P \rangle]\!] &\triangleq \overline{\ulcorner [\![R]\!] \urcorner} \langle\!| \ulcorner [\![P]\!] \urcorner |\!\rangle . \mathbf{0} \mid (\!| \{ \ulcorner [\![R]\!] \urcorner : T, \langle\!| \ulcorner [\![P]\!] \urcorner |\!\rangle : T' \} |\!) \\ [\![\ulcorner R \urcorner (x) . P]\!] &\triangleq \underline{\ulcorner [\![R]\!] \urcorner} (\lambda x) \langle\!| x |\!\rangle . [\![P]\!] \mid (\!| \{ \ulcorner [\![R]\!] \urcorner : T \} |\!) \end{aligned}$$

Lastly, we also need to take the type information into account when concluding channel equivalence, to ensure that two terms with initially dissimilar types cannot become channel equivalent after a substitution. Thus we redefine the entailment rule [CHANEQ$_2$] as follows:

$$[\textsc{ChaneQ}_2] \frac{\Gamma, \Psi \Vdash P_1 \equiv P_2 \qquad \Gamma, \Psi \vdash \ulcorner P_1 \urcorner : T \iff \Gamma, \Psi \vdash \ulcorner P_2 \urcorner : T}{\Gamma, \Psi \Vdash \ulcorner P_1 \urcorner \leftrightarrows \ulcorner P_2 \urcorner}$$

Now we can instantiate the generic type system by defining the instance parameters:

$$[\textsc{Term-1}] \frac{\ulcorner P \urcorner : \langle T, \Gamma' \rangle \in \Psi \qquad \Gamma', \Psi \vdash P}{\Gamma, \Psi \vdash \ulcorner P \urcorner : \langle T, \Gamma' \rangle} \quad [\textsc{Term-2}] \frac{\langle\!| \ulcorner P \urcorner |\!\rangle : \langle T, \Gamma' \rangle \in \Psi \qquad \Gamma', \Psi \vdash P}{\Gamma, \Psi \vdash \langle\!| \ulcorner P \urcorner |\!\rangle : \langle T, \Gamma' \rangle}$$

$$[\textsc{T-ass}] \frac{P : T \in \Psi' \implies T \curvearrowleft \Gamma}{\Gamma, \Psi \vdash (\!| \Psi' |\!)} \quad [\textsc{T-cha}]\ \langle T, \Gamma \rangle \leftrightarrows T \quad [\textsc{T-end}]\ \langle T, \Gamma \rangle \curvearrowleft \Gamma \quad [\textsc{Term-3}] \frac{\Gamma(x) = T}{\Gamma, \Psi \vdash x : T}$$

Note in particular the rules [TERM-1] and [TERM-2]: these rules say that the process within a term must be well-typed w.r.t. the type environment in the second component of its type, and that the process-type pair must be represented in the assertion.

Since we include [CHANEQ$_2$] in order to properly simulate the ρ-calculus, all names that eventually become equal during reduction must have the same type. This amounts to requiring that the programmer must know in advance all the names that will be generated by the program during execution. We have yet to find a type system for the ρ-calculus without this constraint.

6 Conclusions and future work

We have presented a generic type system for higher-order Ψ-calculi, which extends a previous type system for first-order Ψ-calculi. Like its predecessor, type judgements for processes are of the form $\Gamma \vdash P$ and are

given by a fixed set of rules. Terms, assertions and conditions are assumed to form nominal datatypes, and only a few requirements on type rules are imposed.

The generic type system allows us to identify what should be required of type systems for higher-order process calculi that are instances of the Ψ-calculus; these requirements take the form of instance assumptions. Thus it may also yield important insights into the general structure of type systems for higher-order calculi, and it may therefore also be taken as a starting point for developing more advanced type systems for any language that can be shown to be an instance of higher-order Ψ-calculi.

Our type system satisfies a general subject-reduction property and can be instantiated to yield type systems with a notion of channel safety for higher-order calculi such as CHOCS, HOπ and also the ρ-calculus. The latter in particular is interesting, as there is no valid encoding of the ρ-calculus into the π-calculus, and thus we cannot capture higher-order typability in a purely first-order setting. This establishes that our generic type system is richer than first-order type systems. However, typability in the ρ-calculus comes at the cost of necessitating that we include type information directly in the definition of channel equivalence. This amounts to saying that the programmer must know (and specify) in advance the type of all names that will be generated during the course of program evaluation. We do not know whether it is possible to create other (non-trivial) type systems for the ρ-calculus without such a restriction.

There are two important lines of future work in this direction: In [13], Hüttel extends the generic type system to consider more general notions of subtyping and resource awareness, and in [14] he also considers *session types* for psi-calculi. Both of these extensions are formulated for first-order Ψ-calculi only, and they would therefore be relevant to also consider in the higher-order setting.

References

[1] Martín Abadi & Andrew D. Gordon (1999): *A Calculus for Cryptographic Protocols: The Spi Calculus*. *Information and Computation* 148(1), pp. 1–70, doi:10.1006/inco.1998.2740.

[2] Alexander R. Bendixen, Bjarke B. Bojesen, Hans Hüttel & Stian Lybech (2022): *Typing Reflection in Higher-Order Psi-calculi*. Technical Report, Department of Computer Science, Aalborg University. Available at `http://icetcs.ru.is/stian/2022/hopsitypes2022techreport.pdf`.

[3] Jesper Bengtson, Magnus Johansson, Joachim Parrow & Björn Victor (2009): *Psi-calculi: Mobile processes, nominal data, and logic*. In: *2009 24th Annual IEEE Symposium on Logic In Computer Science*, IEEE, pp. 39–48, doi:10.1016/S1571-0661(05)80361-5.

[4] Jesper Bengtson, Magnus Johansson, Joachim Parrow & Björn Victor (2011): *Psi-calculi: a framework for mobile processes with nominal data and logic*. *Logical Methods in Computer Science* Volume 7, Issue 1, doi:10.2168/LMCS-7(1:11)2011. Available at `https://lmcs.episciences.org/696`.

[5] Luís Caires (2007): *Logical Semantics of Types for Concurrency*. In Till Mossakowski, Ugo Montanari & Magne Haveraaen, editors: *Algebra and Coalgebra in Computer Science*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 16–35, doi:10.1006/inco.1994.1093.

[6] Marco Carbone & Sergio Maffeis (2003): *On the Expressive Power of Polyadic Synchronisation in Pi-Calculus*. *Nordic Journal of Computing* 10(2), pp. 70–98, doi:10.1016/S1571-0661(05)80361-5.

[7] Romain Demangeon, Daniel Hirschkoff & Davide Sangiorgi (2010): *Termination in higher-order concurrent calculi*. *The Journal of Logic and Algebraic Programming* 79(7), pp. 550–577, doi:10.1016/j.jlap.2010.07.007. The 20th Nordic Workshop on Programming Theory (NWPT 2008).

[8] Murdoch Gabbay & Andrew Pitts (2002): *A New Approach to Abstract Syntax with Variable Binding*. *Formal Asp. Comput.* 13, pp. 341–363, doi:10.1007/s001650200016.

[9] Philippa Gardner & Lucian Wischik (2000): *Explicit fusions*. In: *International Symposium on Mathematical Foundations of Computer Science*, Springer, pp. 373–382, doi:10.1007/3-540-44612-5_33.

[10] Daniele Gorla (2010): *Towards a unified approach to encodability and separation results for process calculi*. *Information and Computation* 208(9), pp. 1031–1053, doi:10.1016/j.ic.2010.05.002.

[11] Matthew Hennessy & James Riely (2002): *Resource Access Control in Systems of Mobile Agents*. *Information and Computation* 173(1), pp. 82–120, doi:10.1006/inco.2001.3089.

[12] Hans Hüttel (2011): *Typed ψ-calculi*. In: *International Conference on Concurrency Theory*, Springer, pp. 265–279, doi:10.1007/978-3-642-23217-6_18.

[13] Hans Hüttel (2014): *Types for Resources in ψ-calculi*. In Martín Abadi & Alberto Lluch Lafuente, editors: *Trustworthy Global Computing*, Springer International Publishing, Cham, pp. 83–102, doi:10.1007/978-3-319-05119-2_6.

[14] Hans Hüttel (2016): *Binary Session Types for Psi-Calculi*. In Atsushi Igarashi, editor: *Programming Languages and Systems*, Springer International Publishing, Cham, pp. 96–115, doi:10.1007/978-3-319-47958-3_6.

[15] Atsushi Igarashi & Naoki Kobayashi (2004): *A generic type system for the Pi-calculus*. *Theoretical Computer Science* 311(1), pp. 121–163, doi:10.1016/S0304-3975(03)00325-6.

[16] Barbara König (2005): *Analysing input/output-capabilities of mobile processes with a generic type system*. *The Journal of Logic and Algebraic Programming* 63(1), pp. 35–58, doi:10.1016/j.jlap.2004.01.004. Special issue on The pi-calculus.

[17] Stian Lybech (2022): *Encodability and Separation for a Reflective and Higher-Order Language*. *Electronic Proceedings in Theoretical Computer Science* 368, Open Publishing Association, pp. 95–112, doi:10.4204/EPTCS.368.6.

[18] L.G. Meredith & Matthias Radestock (2005): *A Reflective Higher-order Calculus*. *Electronic Notes in Theoretical Computer Science* 141(5), pp. 49 – 67, doi:10.1016/j.entcs.2005.05.016. Proceedings of the Workshop on the Foundations of Interactive Computation (FInCo 2005).

[19] Robin Milner (1993): *The Polyadic π-Calculus: a Tutorial*. In: *Logic and Algebra of Specification*, Springer Berlin Heidelberg, pp. 203–246, doi:10.1007/978-3-642-58041-3_6.

[20] Joachim Parrow (2001): *An introduction to the π-calculus*. In: *Handbook of Process Algebra*, Elsevier, pp. 479–543, doi:10.1016/B978-044482830-9/50026-6.

[21] Joachim Parrow, Johannes Borgström, Palle Raabjerg & Johannes Åman Pohjola (2014): *Higher-order psi-calculi*. *Mathematical Structures in Computer Science* 24(2), doi:10.1017/S0960129513000170.

[22] Benjamin Pierce & Davide Sangiorgi (1993): *Typing and subtyping for mobile processes*. In: *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, IEEE, pp. 376–385, doi:10.1109/LICS.1993.287570.

[23] Davide Sangiorgi (1993): *Expressing mobility in process algebras: first-order and higher-order paradigms*. Ph.D. thesis, University of Edinburgh. Available at `http://hdl.handle.net/1842/6569`.

[24] Davide Sangiorgi (1993): *From π-calculus to higher-order π-calculus — and back*. In M. C. Gaudel & J. P. Jouannaud, editors: *TAPSOFT'93: Theory and Practice of Software Development*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 151–166, doi:10.1007/3-540-56610-4_62.

[25] Davide Sangiorgi & David Walker (2003): *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press.

[26] Bent Thomsen (1989): *A Calculus of Higher Order Communicating Systems*. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL' 89*, POPL'89, ACM Press, New York, NY, USA, pp. 143–154, doi:10.1145/75277.75290.

[27] Bent Thomsen (1993): *Plain CHOCS A Second Generation Calculus for Higher Order Processes*. *Acta Inf.* 30(1), p. 1–59, doi:10.1007/BF01200262.