HILDESHEIMER
INFORMATIK-
BERICHTE

ISSN 0941-3014

Thomas Gehrke
Arend Rensink

Process Creation and Full Sequential Composition in a Name-Passing Calculus

7/97 (May 1997)

UNIVERSITÄT
HILDESHEIM

Dieser Bericht ist
herausgegeben vom

Institut für
Informatik

Postfach 10 13 63
31113 Hildesheim

Process Creation and Full Sequential Composition in a Name-Passing Calculus

Thomas Gehrke and Arend Rensink

Institut für Informatik, Universität Hildesheim*

Abstract. This paper presents a first attempt to formulate a process calculus featuring *process creation* and *sequential composition*, instead of the more usual *parallel composition* and *action prefixing*, in a setting where mobility is achieved by communicating channel names. We discuss the questions of scope and name binding, raised by the interaction of mobility and sequential composition. Substitution of names is integrated as a syntactic operator in the calculus.
Although many aspects necessary to model the combination of process creation, sequential composition and name-passing are taken into account, the setup described in this paper is not yet satisfactory, in that it does not give rise to a useful abstract behavioural equivalence. In particular, the natural notion of bisimulation in our calculus gives rise to a relation that is not a congruence.

Keywords: Process Algebra, Mobility, Sequential Composition, Process Creation.

1 Introduction

Reactive and distributed systems are of increasing importance in theory and practice of computer science. These systems can be described by three characteristics: structure, behaviour and data. For the specification of the first two aspects the formalism of *process algebras* [2, 16, 19, 21] is widely used. Process algebras provide a powerful theory on behavioural preorders and equivalences and allow for formal reasoning on correctness issues, but usually they are weaker on the treatment of data. In order to include the data aspect into system specifications, in the recent years languages like *Concurrent ML* [22], *Facile* [11] and *ProFun* [9] have been developed, which combine the paradigms of process algebras and functional programming languages.

The semantic treatment of such concurrent functional languages is not obvious; some approaches are described in [7, 8, 24]. In this paper, we investigate a direct process algebraic formalisation: we present a calculus that can be used as a semantic foundation for the language *ProFun*. This calculus has to deal with higher-order features, because *ProFun* (like *CML* and *Facile*) allows dynamic change of the linkage structure of systems. For this purpose, adopting the ideas of the π-calculus (see [21]), we define a communication mechanism where in particular *channel names* are passed in communication actions. Sangiorgi has shown that this provides all the necessary expressive power for higher-order programming [23].

In process algebras like the π-calculus, concurrency is usually realised by a binary operator $t|u$, which represents the parallel composition of the processes t and u. On the other hand, the concurrent functional languages mentioned above rather rely on a (unary) operator to create or *spawn* a new process, which then runs concurrently to the remainder of the program.

* Postfach 101363, D–31113 Hildesheim; email: {gehrke,rensink}@informatik.uni-hildesheim.de. Tel. (+49) 5121 883 734, Fax: (+49) 5121 883 768. Research partially supported by the HCM network *EXPRESS* (Expressiveness of Languages for Concurrency); first author supported by the DFG project *EREAS* (Entwurf reaktiver Systeme).

Process creation is used in combination with an operator for the *sequential composition* of subprograms, which again is in contrast to (in fact, a generalisation of) the *action prefix* operator seen in most process algebras. It turns out that especially the combination of communication and sequential composition introduces nontrivial questions of *variable scope*, which we solve in this paper by distinguishing between the binding and scoping of variables.

Independent interest in sequential composition exists from the area of *action refinement*; see, e.g., [12, 13]. Action refinement allows for the stepwise construction of reactive systems. Single communication actions are replaced by process terms, which describe the behaviour of these actions in more detail. The notion of action refinement, in its syntactic interpretation as substitution within terms, calls for sequential composition rather than action prefixing. For example, if in a term $a.b.\mathbf{0}$, the action a should be refined by a term t, there is no obvious way to denote the resulting behaviour $t.b.\mathbf{0}$ without resorting to sequential composition.

The interaction of process creation and sequential composition in the setting of process algebra has been studied before by Baeten and Vaandrager in [3] and by Havelund and Larsen in [14, 15]. Only the latter address higher order features as well, also through name passing. Their solution to the scoping problem, however, is quite restrictive, since they essentially return to action prefixing for input actions, which implies all terms that raise scoping questions are *a priori* ruled out.

The current paper, which is an extended version of [10], describes a first attempt to achieve the goals listed above. However, the approach is less than completely successful, in that an important result claimed in [10] is in fact false: namely, in contrast to what was stated there, bisimulation is *not* a congruence over our language. It is the sequential operator that plays us false.

The remainder of this paper is structured as follows: first we introduce in Section 2 the *basic calculus*, containing communication but no mobility; we give an operational semantics and a complete axiomatisation for finite terms. The *full calculus*, extending the basic calculus with the possibility to communicate channel names, is presented in Section 3, again with operational treatment. Finally, Section 4 compares the approaches mentioned above and contains some concluding remarks. The proofs of the theorems and propositions can be found in the appendix.

2 The Basic Calculus

In this chapter we introduce the basic calculus. In contrast to the full calculus, it does not allow parameter passing during communication or process invocation.

Syntax and operational semantics. Similar to CCS [19] we consider communication to be a synchronous action between two processes which can perform corresponding communication actions. We assume a countable set $\mathcal{C}$ of channel names, ranged over by a,b,c. A channel a can be used either for *input*, denoted $a?$, or for *output*, denoted $a!$. We sometimes use '$\dagger$' to as a "metavariable" denoting either '!' or '?'. The set of communication actions is denoted $\mathcal{A} = \{a\dagger \mid a \in \mathcal{C}\}$, ranged over by α, β. We represent internal behaviour by $\tau = \iota!$, where $\iota \in \mathcal{C}$ is a special channel which may not otherwise occur. $\mathcal{N}$, ranged over by n, n', is a set of process names. The basic calculus of this section, $\mathcal{B}$, ranged over by t, u, v, is defined through the following grammar:

$$t ::= \mathbf{1} \mid g \mid \mathit{spawn}(t) \mid (a : t) \mid t;t \ .$$

$$g ::= \mathbf{0} \mid g + g \mid (a : g) \mid g; t \mid t; g \mid n \mid \alpha \ .$$

We distinguish between *guarded terms* (g) and *non-guarded terms* (t), where the former start with an action before they may terminate. $\mathbf{1}$ denotes a successfully terminated term, $\mathbf{0}$ the inactive process. $spawn(t)$ creates a new process which performs t concurrently to the spawning term. $(a : t)$ restricts the execution of t to the actions in $\mathcal{A}\backslash\{a?, a!\}$. $+$ is the guarded choice operator, which is resolved by the execution of one of its alternatives. $t; u$ denotes the sequential composition of t and u, i.e., u can perform actions when t has terminated. A process name $n \in \mathcal{N}$ is interpreted by a function $\Theta : \mathcal{N} \to \mathcal{B}$: n denotes a process call of $\Theta(n)$. The unfolding of the definition will accompanied by an internal action, hence such a call is guarded. For syntactical convenience we assume that ; has a higher priority than $+$; for instance, $a! + b!; c!$ is $a! + (b!; c!)$. Furthermore, we assume sequential composition to be right associative, i.e. $t; u; v$ is $t; (u; v)$.

Now we define the operational semantics of the basic calculus. For this purpose, we use the general notion of a *labelled transition system* [19], extended with a predicate to denote the *successful termination* of a state.

Definition 1. A labelled $\checkmark$-transition system is a tuple $\langle \mathcal{L}, S, \to, \checkmark \rangle$ where
- $\mathcal{L}$ is a *label set*;
- S is a set of *states*;
- $\to \subseteq S \times \mathcal{L} \times S$ is a *transition relation*, whose elements are denoted $s \xrightarrow{\ell} s'$;
- $\checkmark \subseteq S$ is a *termination predicate*, such that $s \in \checkmark$ and $s \xrightarrow{\ell} s'$ implies $s' \in \checkmark$.

Transition systems are ranged over by T, U. For the calculus presented above, we have $\mathcal{L} = \mathcal{A}$ and $S = \mathcal{B}$. For $s \in \checkmark$ we write $s\checkmark$. The termination and transition predicates are defined through operational rules, in Figure 1.

The termination predicate extends the usual notion, in that terminated terms may at the same time still perform actions, namely if they are *spawn*ed off as parallel processes. (A similar approach is seen in [3].) Note that we need no rule for the termination of choice, since the restriction to *guarded* choice guarantees that in $t + u$, neither t nor u can be terminated. This simplifies matters greatly and is, in fact, precisely the reason for the restriction to guarded choice. Since there appears to be growing consensus that guarded choice suffices in practical applications of process calculi, our restriction seems quite reasonable.

With respect to sequential composition, the standard operational rules are as follows (cf. [2]):

$$\frac{t \xrightarrow{\alpha} t'}{t; u \xrightarrow{\alpha} t'; u} \qquad \frac{t\checkmark \quad u \xrightarrow{\alpha} u'}{t; u \xrightarrow{\alpha} u'}$$

In our setup, the first rule is fine but the second one is not, since it discards the first operand. In the case where the first operand equals $spawn(t)$ for some t, this is not the desired behaviour; rather, $spawn(t)$ should still be there in the target term. In general, if the first operand is terminated, the sequential composition behaves very much like standard *parallel* composition. This is indeed our intuition; in fact we also allow *communication* between $spawn(t)$ and u in $spawn(t); u$.

Apart from sequential composition, there is only one unusual rule in our semantics, namely the one for recursion, which specifies an internal step to perform the unfolding of a process call into its body. Our new approach to sequential composition is realised in the rule R_6 for *spawn* and the three rules R_7, R_8 and R_9 for sequential

$$\frac{}{\mathbf{1}\checkmark}\mathrm{T}_1 \qquad \frac{}{spawn(t)\checkmark}\mathrm{T}_2 \qquad \frac{t\checkmark \quad u\checkmark}{(t;u)\checkmark}\mathrm{T}_3 \qquad \frac{t\checkmark}{(a:t)\checkmark}\mathrm{T}_4$$

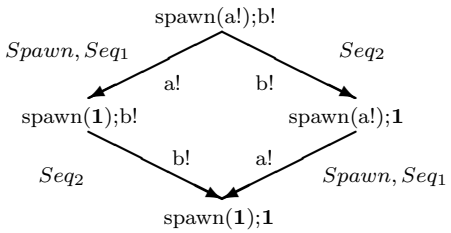
$$\frac{}{\alpha \xrightarrow{\alpha} \mathbf{1}}\mathrm{R}_1 \qquad \frac{}{n \xrightarrow{\tau} \Theta(n)}\mathrm{R}_2 \qquad \frac{t \xrightarrow{\alpha} t'}{t+u \xrightarrow{\alpha} t'}\mathrm{R}_3 \qquad \frac{u \xrightarrow{\alpha} u'}{t+u \xrightarrow{\alpha} u'}\mathrm{R}_4$$

$$\frac{t \xrightarrow{a\dagger} t' \quad a \neq b}{(b:t) \xrightarrow{a\dagger} (b:t')}\mathrm{R}_5 \qquad \frac{t \xrightarrow{\alpha} t'}{spawn(t) \xrightarrow{\alpha} spawn(t')}\mathrm{R}_6$$

$$\frac{t \xrightarrow{\alpha} t'}{t;u \xrightarrow{\alpha} t';u}\mathrm{R}_7 \qquad \frac{t\checkmark \quad u \xrightarrow{\alpha} u'}{t;u \xrightarrow{\alpha} t;u'}\mathrm{R}_8 \qquad \frac{t\checkmark \quad t \xrightarrow{\alpha} t' \quad u \xrightarrow{\beta} u' \quad \{\alpha,\beta\} = \{a!,a?\}}{t;u \xrightarrow{\tau} t';u'}\mathrm{R}_9$$

Fig. 1. Transition rules for the basic calculus.

composition. In particular, R_9 expresses communication. If a process term t is terminated, but may also perform an action α, it is clear that t must contain a term of the form $spawn(t'')$. If u is able to perform the corresponding action β such that $\{\alpha,\beta\} = \{a!,a?\}$, implying that α and β specify input and output over the same channel, communication is possible. Consider the following example:



The operational semantics generates a ✓-transition system. In particular, the condition regarding the persistency of termination is satisfied.

Proposition 2. *$\langle \mathcal{A}, \mathcal{B}, \rightarrow, \checkmark \rangle$ is a ✓-transition system.*

Bisimulation and Axiomatisation. We define a notion of process equivalence which is based on the concept of *bisimulation* [19]. Two processes are called *bisimilar* if it is not possible for an external observer to distinguish between their behaviours. We will treat the internal action τ just like any other action. This leads to a rather strict equivalence which is called *strong bisimulation*.

Definition 3. Let T be a ✓-transition system. A symmetrical relation $R \subseteq S \times S$ is called a *bisimulation relation* if for all $(s_1, s_2) \in R$
- if $s_1 \xrightarrow{\ell} s_1'$ then $\exists s_2 \xrightarrow{\ell} s_2'$ such that $(s_1', s_2') \in R$;
- if $s_1\checkmark$ then $s_2\checkmark$.

$s_1, s_2 \in S$ are called *bisimilar*, denoted $s_1 \sim_{\mathcal{B}} s_2$, if $(s_1, s_2) \in R$ for some bisimulation relation R.

The only non-standard part is the condition on the termination predicates, which is necessary to ensure congruence. Without this condition, we would have $spawn(a!) \sim_{\mathcal{B}} a!$, but these terms generate different behaviour in the context of sequential composition: for instance, $spawn(a!);b! \xrightarrow{b!} spawn(a!);\mathbf{1}$ whereas $a!;b! \not\xrightarrow{b!}$.

We establish that $\sim_{\mathcal{B}}$ is a congruence w.r.t. the operators of our calculus. For this purpose, rather than giving a direct proof, we derive the result from existing

$$\mathbf{1};t = t \quad (1) \qquad t;\mathbf{1} = t \quad (2) \qquad \mathbf{0};t = \mathbf{0} \quad (3)$$
$$t;(u;v) = (t;u);v \quad (4) \qquad t + \mathbf{0} = t \quad (5) \qquad t + u = u + t \quad (6)$$
$$t + t = t \quad (7) \qquad (t + u) + v = t + (u + v) \quad (8) \qquad (t + u);v = t;v + u;v \quad (9)$$
$$n = \tau;\Theta(n) \quad (10) \qquad (a : \mathbf{0}) = \mathbf{0} \quad (11) \qquad (a : \mathbf{1}) = \mathbf{1} \quad (12)$$
$$(a : (a : t)) = (a : t) \quad (13) \qquad (a : (b : t)) = (b : (a : t)) \quad (14)$$
$$(a : a\dagger;t) = \mathbf{0} \quad (15) \qquad (a : b\dagger;t) = b\dagger;(a : t) \quad \text{if } a \neq b \quad (16)$$
$$(a : t + u) = (a : t) + (a : u) \quad (17) \qquad (a : spawn(t)) = spawn((a : t)) \quad (18)$$
$$spawn(\mathbf{0}) = \mathbf{1} \quad (19)$$
$$spawn(t);spawn(u) = spawn(u);spawn(t) \quad (20)$$
$$spawn(t);spawn(u) = spawn(spawn(t);u) \quad (21)$$
$$spawn(t;spawn(u) + v) = spawn(t;u + v) \quad (22)$$
$$\text{if } t \equiv \sum_{i\in\mathcal{I}} \alpha_i;t_i \text{ and } u \equiv \sum_{k\in\mathcal{K}} \beta_k;u_k \text{ then} \quad (23)$$
$$spawn(t);u = \sum_{i\in\mathcal{I}} \alpha_i;spawn(t_i);u + \sum_{k\in\mathcal{K}} \beta_k;spawn(t);u_k + \sum_{\substack{i\in\mathcal{I}\\ k\in\mathcal{K}\\ \{\alpha_i,\beta_k\}=\{a?,a!\}}} \tau;spawn(t_i);u_k$$

Fig. 2. Axioms of the basic calculus.

$$spawn(\mathbf{1}) = \mathbf{1} \quad (24)$$
$$spawn(spawn(t)) = spawn(t) \quad (25)$$
$$spawn(t;spawn(u)) = spawn(t;u) \quad (26)$$

Fig. 3. Derived equations.

meta-theory. It has been shown that, if the rules of the operational semantics are compatible with certain *formats*, we are able to establish properties of this semantics. A typical property which can be proved in this way is the congruence of equivalence relations [5, 17]. Because of the occurrence of the predicate ✓ in our rules we have to use the *path* format [1] which allows the use of predicates. It is easy to verify that all the rules in Figure 1 satisfy the conditions for the *path* format, so (strong) bisimulation is a congruence for our calculus.

Theorem 4. *$\sim_{\mathcal{B}}$ is a congruence over the basic calculus.*

In Figure 2, we give a set $\mathcal{AX}_{\mathcal{B}}$ of axioms for the axiomatisation of $\sim_{\mathcal{B}}$. Examples for derived equations are given in Figure 3.

Theorem 5. *The theory $\mathcal{AX}_{\mathcal{B}}$ is sound with respect to $\sim_{\mathcal{B}}$.*

We denote the finite fragment of $\mathcal{B}$, i.e., without the recursion operator, by $\mathcal{B}_{fin}$. For the proof of completeness it is useful to define *normal forms* of terms. Therefore, we use the sum notation for a more concise representation of choice operators: if $I = \{i_1, \ldots, i_n\}$ then $\sum_{i\in I} t_i = t_{i_1} + ... + t_{i_n}$, where $\sum_{i\in\emptyset} t_i$ equals $\mathbf{0}$ and $\sum_{i\in\{x\}} t_i$ equals t_x. This is a valid notation because of Eqs. (5)–(8).

Definition 6. A term $t \in \mathcal{B}_{fin}$ is in *basic normal form* if t is a term in N:

$$N ::= \sum_{i\in\mathcal{I}} \alpha_i;N_i \mid spawn(B) \qquad B ::= \sum_{i\in\mathcal{I}} \alpha_i;B_i$$

A term is in simple basic normal form if it is a term in B.

Normal form terms do not contain nested *spawn* applications (hence, for instance, the term $\mathit{spawn}(a!; \mathit{spawn}(b!) + c!; \mathit{spawn}(d!)); e!$ is not in normal form), use only action prefix and no restriction operators. This means that basic normal form terms abstract from the individual *spawn*-applications and just describe the possible interleaving sequences of actions a term can perform. For instance, a term α has the basic normal form $\alpha; \mathit{spawn}(\mathbf{0})$. For another example, by the expansion law (Equation (23)) and other axioms it can be deduced that

$$\begin{aligned}\mathit{spawn}(a!; \mathit{spawn}(b!) + c!; \mathit{spawn}(d!)); c? = a!; (b!; c? + c?; \mathit{spawn}(b!))\\ + c!; (d!; c? + c?; \mathit{spawn}(d!)) + c?; \mathit{spawn}(a!; b! + c!; d!) + \tau; \mathit{spawn}(d!)\end{aligned}$$

With the existence of a normal form of each term $t \in \mathcal{B}_{\mathit{fin}}$, we can deduce a completeness result of $\mathcal{AX}_{\mathcal{B}}$ for finite process terms, i.e., terms not containing process calls.

Theorem 7. *For all $t \in \mathcal{B}_{\mathit{fin}}$, there is an $u \in \mathcal{B}_{\mathit{fin}}$ in basic normal form such that $\mathcal{AX}_{\mathcal{B}} \vdash t = u$.*

Theorem 8. *The theory $\mathcal{AX}_{\mathcal{B}}$ is complete for $\sim_{\mathcal{B}}$ on the finite fragment $\mathcal{B}_{\mathit{fin}}$ of $\mathcal{B}$.*

3 The Full Calculus

We now extend the basic calculus with mobility in the fashion of the π-calculus. It turns out that due to the presence of sequential composition in the language, some of the assumptions underlying the π-calculus have to be reconsidered. At the end of the section, we show that the current setup cannot be the last word, since it gives rise to a notion of bisimulation that is not a congruence w.r.t. sequential composition.

The basic question is one of binding and scope. For instance, in the term $t_1 = (x?y; y!z); z!y$, the second occurrence of y is clearly bound by the first; but what about the third? Since we want to preserve associativity of sequential composition, the answer is immediate: in $x?y; (y!z; z!y)$, both of the latter y's are bound by the first, hence this must be the case in t_1 as well.

As a further step, consider $t_2 = (x?z + y?z); z!a$. Here, it is not uniquely determined what the binding occurrence of the latter z is; depending on how the choice is resolved, it could be either of the first two z's. One might argue that terms with this property should be disallowed; however, we feel that t_2 is a typical example why sequential composition is considered practically useful. Since sequential composition right-distributes over choice, t_2 is equivalent to $t_2' = x?z; z!a + y?z; z!a$; however, t_2' does not immediately convey the fact that the choice operands differ only in the first action. In fact, it turns out that terms like t_2 pose no essential complication in the theory.

In terms such as $t_3 = (x?y + x?z); z!a$, the question whether the second z is bound at all appears to depend on the resolution of the choice. However, the property of unique binding (a variable receives a value only once) is necessary for a smooth formalisation of the semantics; therefore every variable should be either bound or free in a given term. For this reason we say that in the left hand operand of the subterm $x?y + x?z$ of t_3, z is implicitly bound, viz. to itself.

A further complication, also due to sequential composition, lies in the notion of syntactic substitution, which is the basic mechanism for replacing variables by values in the π-calculus. In our calculus, the scope of a bound variable is in general unlimited, except when explicitly restricted. For instance, $x?y$ binds y in all subsequent

subterms; as long as no explicit scope restriction is given, it will always be possible to sequentially append further y-containing terms. This is in contrast to the action prefix term $x?y;t$, where y is only valid within the given term t. As a consequence, in our calculus it is not a priori clear where to apply substitution and where to stop applying it.

Restriction. We solve these problems by distinguishing between variable binding and scoping. A variable x can be bound explicitly through a receive action $(a?x)$, or implicitly, corresponding to the dynamic generation of a new channel. Scope restriction is denoted $(x : t)$ as before. As mentioned above, we keep to the declarative principle that a variable is bound, i.e., receives a value, only once in its lifetime. On the other hand, it is also restricted only once in its lifetime.

The operator $(x : t)$ has twofold effect. First, it restricts the scope of x to the term t. Second, it influences the syntax of the context of t, because in a term $t' = u;(x : t);v$ the name x must not occur in u or v; otherwise the t' would not be *well-formed* (see below).

With respect to the scoping aspects of restriction, a phenomenon occurs in the operational semantics that is known from the π-calculus: the scope of a channel name can change during the lifetime of a system. The situation that a channel name becomes known outside its original scope is called *scope extrusion*. It is reflected by a syntactic change: the restriction operator disappears, and is subsequently reapplied to a super-term of its original operand. For a proper treatment of this phenomenon, we rely on a notion of *restricted names* (corresponding to the π-calculus' *bound* names). For every channel name $a \in \mathcal{C}$, we assume a restricted name $\tilde{a}$. We define $\tilde{\mathcal{C}} = \{\tilde{a} \mid a \in \mathcal{C}\}$, and use χ, ξ to range over $\mathcal{C} \cup \tilde{\mathcal{C}}$. For every $\Xi \subseteq \mathcal{C} \cup \tilde{\mathcal{C}}$, combining restricted and unrestricted channels, $r_\Xi = \{a \mid \tilde{a} \in \Xi\}$ denotes the "restriction content" of Ξ, and $c_\Xi = (\Xi \cap \mathcal{C}) \cup r_\Xi$ denotes the original channel names in Ξ.

Summarising, we have three kinds of variable occurrence: variables can be free (visible and unassigned), bound (visible, but with an assigned, though as yet unknown, value) and restricted (invisible). Binding occurs *implicitly* when restricting a non-bound variable, and when specifying a choice between operands with distinct sets of bound variables. Implicit binding always generates a *fresh* value, which is syntactically indicated by the variable name itself.

Syntactic substitution. We need a way to connect concrete values to variables. Again following the ideas of the π-calculus, we use a notion of substitution for this purpose. However, as discussed above, the scoping aspects of sequential composition require a more sophisticated approach than in the case of action prefixing; in fact, we need a form of "delayed" substitution, which is stored for future use. For this purpose, we introduce substitutions as part of the syntax of our calculus.

In general, substitution will be finite sets $\sigma = \{x_1 \leftarrow \xi_1, \ldots, x_m \leftarrow \xi_m\}$, where $x_i \in \mathcal{C}$ and $\xi_i \in \mathcal{C} \cup \tilde{\mathcal{C}}$ such that $x_i \neq c_{\xi_i}$, $x_i \neq x_j$ for all distinct i, j, and $c_{\xi_i} = c_{\xi_j}$ implies $\xi_i = \xi_j$ (hence images with the same channel name have the same restriction content). We write $\operatorname{dom}\sigma = \{x_1, \ldots, x_m\}$ for the domain of σ, $\operatorname{rng}\sigma = \{\xi_1, \ldots, \xi_m\}$ for its range, $\sigma(x_i) = \xi_i$ for all $1 \leq i \leq m$ (which is well-defined due to the above requirements on σ) and $\sigma(x) = x$ for all $x \notin \operatorname{dom}\sigma$ (hence σ may be considered as a function $\mathcal{C} \to (\mathcal{C} \cup \tilde{\mathcal{C}})$). The class of substitutions is denoted $\mathbf{S}$.

A substitution σ indicates that all $x \in \operatorname{dom}\sigma$ are to be bound to the corresponding channel $c_{\sigma(x)}$, while at the same time $r_{\sigma(x)}$ is to be restricted, to deal with scope

extrusion. We define the following constructions on substitutions:

$$\begin{aligned}
\sigma \uparrow a &= \{(x,a) \mid (x,\tilde{a}) \in \sigma\} \cup \{(x,\xi) \in \sigma \mid a \neq c_\xi\} \\
\sigma \downarrow a &= \{(x,\tilde{a}) \mid (x,a) \in \sigma\} \cup \{(x,\xi) \in \sigma \mid x \neq a \neq c_\xi\} \\
\sigma_1 \circ \sigma_2 &= \{(x,\sigma_1(\xi)) \mid (x,\xi) \in \sigma_2, \sigma_1(\xi) \neq x\} \cup \{(x,\xi) \in \sigma_1 \mid x \notin \operatorname{dom}\sigma_2\}
\end{aligned}$$

$\sigma \uparrow a$ "frees" the image a in σ, i.e., changes it from restricted to unrestricted; the dual construction $\sigma \downarrow a$ changes it into restricted, and also removes a from $\operatorname{dom}\sigma$ (if it was there). Finally, $\sigma_1 \circ \sigma_2$ is the composition of the substitutions, considered as (partial) functions.

σ is called *free* (that is, non-restricted) if $\operatorname{rng}\sigma \subseteq \mathcal{C}$. Free substitutions are used as explicit language constants; non-free substitutions only occur as part of the semantics. We sometimes write a free $\sigma = \{x_1{\leftarrow}a_1, \ldots, x_m{\leftarrow}a_m\}$ as $\mathbf{x}{\leftarrow}\mathbf{a}$, where the $\mathbf{x}$ and $\mathbf{a}$ represent *vectors* of channels corresponding to $x_1 \cdots x_m$ and $a_1 \cdots a_m$, respectively. $\{\mathbf{x}\} = \{x_1, \ldots, x_m\}$ denotes the set of elements of the vector $\mathbf{x}$. We also write $\sigma(\mathbf{y})$, with the obvious meaning.

The problems of scope and binding are aggravated by the introduction of syntactic substitution, because in combination with restriction and sequential composition it gives rise to a new form of scope extrusion, which could be called *forward extrusion* in contrast to the known *parallel extrusion* through communication. For instance, the term $(a : \{x{\leftarrow}a\}; t); x!b$ expresses that all x are to be replaced by a, including the last x, which is outside the a-restriction; hence a becomes known outside its scope. Rather than giving $(a : t'); \{x{\leftarrow}a\}; x!b$ as the result of this substitution, we extend the restriction to cover $x!b$, i.e., the result of the substitution is equivalent to $(a : \{x{\leftarrow}a\}; t; x!b)$.

The full calculus, denoted $\mathcal{F}$, is generated by the following grammar:

$$\begin{aligned}
t &::= \mathbf{1} \mid \sigma \mid g \mid \mathit{spawn}(t) \mid (x : t) \mid t; t \ . \\
g &::= \mathbf{0} \mid g + g \mid (x : g) \mid g; t \mid t; g \mid n(\mathbf{x}) \mid x!x \mid x?x \mid [x{=}x] \ .
\end{aligned}$$

As before, t stands for an arbitrary term and g for a guarded term; x stands for a channel variable, and σ for a free substitution. (Note that restricted channel names cannot occur anywhere in the syntax.) $[a{=}b]$ corresponds to the *matching* operator of the π-calculus: if $a = b$ then it is equivalent to τ, otherwise to $\mathbf{0}$. We sometimes use $(\mathbf{x} : t)$ to abbreviate $(x_1 : (x_2 : \cdots (x_m : t) \cdots))$. The process environment Θ is assumed to consist of rules of the form $n(\mathbf{x}) \mapsto t$, where $\mathbf{x}$ is a vector of formal parameters. Such rules are *interpreted up to α-conversion*, meaning that the names in $\mathbf{x}$, as well as other variable names local to t, can be replaced by arbitrary different names. This is necessary because semantically, a process call $n(\mathbf{a})$ is treated by "inlining" a substitution instance of its body t; this could give rise to a non-well-formed term (see below) if variable names in t cannot be chosen at will.[2]

Well-formed terms. Not all terms of $\mathcal{F}$ are acceptable; for instance, as discussed above, we want unique binding of variables. More precisely, we want the following informal properties to be satisfied:

– No variable is bound sequentially *after* it occurs free.

[2] Note that α-conversion is *different* from the notion of substitution regarded here, since the former also replaces restricting and binding occurrences of variables.

t	$fv(t)$	$bv(t)$	$rv(t)$
$\mathbf{0}$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathbf{1}$	$\emptyset$	$\emptyset$	$\emptyset$
σ	$\operatorname{dom}\sigma \cup \operatorname{rng}\sigma$	$\emptyset$	$\emptyset$
$n(\mathbf{a})$	$\{\mathbf{a}\}$	$\emptyset$	$\emptyset$
$x!y$	$\{x, y\}$	$\emptyset$	$\emptyset$
$x?y$	$\{x\}$	$\{y\}$	$\emptyset$
$[x{=}y]$	$\{x, y\}$	$\emptyset$	$\emptyset$
$u + v$	$(fv(u) \setminus bv(v)) \cup (fv(v) \setminus bv(u))$	$bv(u) \cup bv(v)$	$rv(u) \cup rv(v)$
$(x : u)$	$fv(u) \setminus \{x\}$	$bv(u) \setminus \{x\}$	$rv(u) \cup \{x\}$
$spawn(u)$	$fv(u)$	$\emptyset$	$rv(u)$
$u; v$	$fv(u) \cup (fv(v) \setminus bv(u))$	$bv(u) \cup bv(v)$	$rv(u) \cup rv(v)$

Fig. 4. Free, bound and restricted variables.

- Variables bound within a *spawn* or process definition should be restricted to that scope. This condition ensures the locality of binding.
- Restricted variables may not occur outside their scope.

This is formalised using the concepts of *free*, *bound* and *restricted* variables of a term t, defined in Figure 4 as $fv(t)$, $bv(t)$ and $rv(t)$, respectively. We also use $var(t) = fv(t) \cup bv(t) \cup rv(t)$.

Example 1. Consider the term $t = b?a + a!c$. In the left hand operand, a is bound by communication. With the rules for choice we can deduce that $fv(t) = \{b, c\}$ and $bv(t) = \{a\}$, therefore a is implicitly bound in the right hand operand; its value is assumed to be a itself. a is even bound implicitly in terms like $b?a + c!d$, where it does not occur in the other alternative. Implicit binding also takes place in restriction operators: in $t = (a : x!a)$, the name a is bound implicitly.

The purpose of this definition is to restrict the set of allowable terms; in the remainder we will assume that the following conditions are satisfied:

- for all terms $(x : t)$, $x \notin rv(t)$;
- for all terms $t; u$, $rv(t) \cap var(u) = var(t) \cap (bv(u) \cup rv(u)) = \emptyset$;
- for all terms $spawn(t)$, $bv(t) = \emptyset$
- for all definitions $\Theta{:}\, n(\mathbf{x}) \mapsto t$, $bv(t) = \emptyset$ and $fv(t) \subseteq \{x_1, \ldots, x_m\}$.

The first two conditions ensure that each name is restricted at most once. The third condition demands that each name bound in a spawned process must be restricted. The fourth condition ensures that in a process definition, all bound names have to be restricted as well, and the free names have to be a subset of the parameters. The latter two conditions realise the concept of *locality* known from programming languages. Terms satisfying these conditions are called *well-formed*. In the remainder of the paper, we implicitly restrict to well-formed terms, unless stated otherwise.

Example 2. The following terms are not well-formed: $(a : a!b); a!c$, $spawn(a?b)$, $(a : b!a) + a?c$, $b?a; c?a$, $\Theta : n(x) \mapsto a!x$.

Structural equivalence. The effect of substitution is not expressed operationally. Instead, we adapt the idea of a *structural equivalence*, proposed for another purpose by Milner in [20], to capture the effect of substitution. $\equiv$ is defined as the smallest congruence satisfying the equations in Figure 5.

$\mathbf{1}; t = t$	(27)	$\sigma; \mathbf{0} = \mathbf{0}; \sigma$	(34)
$t; \emptyset = t$	(28)	$\sigma; \sigma' = \sigma \circ \sigma'$	(35)
$(t; u); v = t; (u; v)$	(29)	$\sigma; x \dagger y = \sigma(x) \dagger \sigma(y); \sigma$	(36)
$(x : (y : t)) = (y : (x : t))$	(30)	$\sigma; [x{=}y] = [\sigma(x){=}\sigma(y)]; \sigma$	(37)
$t; (x : u) = (x : t; u)$	(31)	$\sigma; n(\mathbf{x}) = n(\sigma(\mathbf{x})); \sigma$	(38)
$(x : t); u = (x : t; u)$	(32)	$\sigma; \mathit{spawn}(t) = \mathit{spawn}(\sigma; t); \sigma$	(39)
$(x : t; \sigma) = t; \sigma \setminus \{x \leftarrow a\}$ if $x \leftarrow a \in \sigma, x \notin \operatorname{rng} \sigma, x \notin \mathit{var}(t)$	(33)	$\sigma; (t + u) = \sigma; t + \sigma; u$	(40)

Fig. 5. Structural equivalence

Note that the restriction to well-formed terms drastically limits the applicability of the structural equivalence axioms. For instance, by applying (30)–(32), we can derive $(b, x : a?c; x!b; x!c) \equiv a?c; (x : (b : x!b); x!c)$ but not $(x : (b : x!b); x!c) \equiv (x, b : x!b); x!c$, since the latter term is not well-formed. On the other hand, the axioms (31) and (32) can always be applied from left to right, in which case they precisely describe the principle of scope extrusion. The precise (technical) requirement for structural equivalence lies in a special syntactic form that all terms can be rewritten to.

Definition 9. A term is in *structural normal form* (*snf*) if it equals $(\mathbf{x} : t_1; \cdots; t_n; \sigma)$ for some $n \geq 0$, where $\{\mathbf{x}\} \cap \operatorname{dom} \sigma = \emptyset$ and for all $1 \leq i \leq n$, t_i equals one of the following:

- $a!b$, $a?b$, $[x{=}y]$, $n(\mathbf{a})$, $\mathbf{1}$ or $\mathbf{0}$;
- $\mathit{spawn}(t_i')$ where t_i' is in *snf*;
- $\sum_{k \in K_i} u_k$ where $|K_i| > 1$ and for all $k \in K_i$, u_k is in *snf*.

We then have the following property:

Proposition 10. *For every $t \in \mathcal{F}$, there is a unique $u \equiv t$ with u in* snf.

Termination. We have seen above that the effect of substitution is not restricted to the term currently in question, but may also extend to its context, in particular to sequentially appended terms. Furthermore, the effect of substitution may be modified by scope extrusion. For instance, the term $(a : \{x \leftarrow a\})$ not only has the substitution $x \leftarrow a$ that may carry over to the right, as in $(a : \{x \leftarrow a\}); b!x$, but also the restricted name a that may escape its current scope by this means.

Operationally, we deal with this effect by adapting the termination predicate, so that it records additional information about the "residual" of a terminated term, consisting of the remaining substitution and the resultant scope extrusion. Residuals are modelled as (non-free) substitution functions. For the full calculus, therefore, termination will be modelled by an indexed predicate $\checkmark_\sigma$, where σ is a substitution, and $r_{\operatorname{rng} \sigma}$ expresses which of the σ-images are scope-extruded by substitution. For instance, $(a : \{x \leftarrow a\})\checkmark_{\{x \leftarrow \tilde{a}\}}$. The special case $\checkmark_\emptyset$ is abbreviated to $\checkmark$. The rules of termination are listed in Figure 6.

Operational semantics. The transition rules for the full calculus extend those of the basic calculus with channel parameters. Transition labels are the following:

- $a!b$: output of value b over channel a;
- $a!\tilde{b}$: output of a *fresh* value b over a (called *restricted output*);
- $a?x$: input of a value over channel a, to be assigned to the variable x;
- $a?\tilde{x}$: *restricted* input over a, to be assigned to the *local* variable x.

$$\frac{}{\mathbf{1}\checkmark}\,\mathrm{T}_5 \qquad \frac{}{spawn(t)\checkmark}\,\mathrm{T}_6 \qquad \frac{}{\sigma\checkmark_\sigma}\,\mathrm{T}_7 \qquad \frac{t\checkmark_{\sigma_t} \quad u\checkmark_{\sigma_u}}{(t;u)\checkmark_{\sigma_t\circ\sigma_u}}\,\mathrm{T}_8 \qquad \frac{t\checkmark_\sigma}{(a:t)\checkmark_{\sigma\downarrow a}}\,\mathrm{T}_9 \qquad \frac{t\checkmark_\sigma \quad u\equiv t}{u\checkmark_\sigma}\,\mathrm{T}_{10}$$

$$\frac{}{a!b \xrightarrow{a!b} \mathbf{1}}\,\mathrm{R}_{10} \qquad \frac{}{a?x \xrightarrow{a?x} \mathbf{1}}\,\mathrm{R}_{11} \qquad \frac{}{[a{=}a] \xrightarrow{\tau} \mathbf{1}}\,\mathrm{R}_{12} \qquad \frac{\Theta{:}\, n(\mathbf{x}) \mapsto t}{n(\mathbf{a}) \xrightarrow{\tau} (\mathbf{x} : \mathbf{x}{\leftarrow}\mathbf{a}; t)}\,\mathrm{R}_{13}$$

$$\frac{t \xrightarrow{a\dagger\xi} t' \quad a \neq b \neq c_\xi}{(b:t) \xrightarrow{a\dagger\xi} (b:t')}\,\mathrm{R}_{14} \qquad \frac{t \xrightarrow{a\dagger b} t' \quad a \neq b}{(b:t) \xrightarrow{a\dagger\tilde{b}} t'}\,\mathrm{R}_{15}$$

$$\frac{t\checkmark \quad t \xrightarrow{\alpha} t' \quad u \xrightarrow{\beta} u' \quad \{\alpha,\beta\} = \{a!\xi, a?\chi\}}{t;u \xrightarrow{\tau} (r_{\{\xi,\chi\}} : \{c_\chi{\leftarrow}c_\xi\}; t'; u')}\,\mathrm{R}_{16} \qquad \frac{u \equiv t \quad t \xrightarrow{\alpha} t' \quad t' \equiv u'}{u \xrightarrow{\alpha} u'}\,\mathrm{R}_{17}$$

Fig. 6. Transition rules for the full calculus

Again, internal action labels are treated as a special case of output: $\tau = \iota!\iota$, where $\iota \notin bv(t) \cup rv(t)$ for all terms t. The set of transition labels is denoted $\mathcal{L}$, ranged over by α, β. Restricted input and output are generated when input or output actions occur within a scope restriction.

The definition of a transition system has to be adapted to the extended termination predicate and transition labels. Namely, if $s \xrightarrow{a\dagger\tilde{b}} s'$ and s is terminated with residual substitution σ, then s' is terminated, too, such that in the corresponding residual, the b-images of σ are "freed" by scope extrusion (see also rule R_{15}).

Definition 11. An extended $\checkmark$-transition system is a tuple $\langle \mathcal{L}, S, \rightarrow, \checkmark \rangle$ where $\checkmark \subseteq S \times \mathbf{S}$ is a termination relation, such that if $s\checkmark_\sigma$ and $s \xrightarrow{a\dagger\xi} s'$ then $s'\checkmark_{\sigma\uparrow c_\xi}$.

The operational rules for the choice and *spawn* operators and the non-communication rules of sequential composition are unchanged, and omitted here. For the other operators, the rules are given in Figure 6. Some comments on the operational rules are in order.

- The rule R_{13} for recursion inserts a substitution in front of the term $\Theta(n)$, which replaces the formal parameters by the current names in the process call. Additionally, the formal parameters are restricted to the term t to ensure their locality.
- The rule R_{16} for communication combines a number of features. Two labels α and β can communicate if and only if $\{\alpha, \beta\} = \{a!\xi, a?\chi\}$ for some a, ξ, χ; this results in a syntactic substitution $c_\chi{\leftarrow}c_\xi$ which implements the transfer of a data value (i.e., a channel name), and a potential restriction of $r_{\{\xi,\chi\}}$ $(= r_\xi \cup r_\chi)$, which implements scope extrusion. (Note that r_ξ is non-empty iff $a!\xi$ is a restricted output, and r_χ iff $a?\chi$ a restricted input.)
  The communication rule corresponds to *late binding*. For instance, we can derive

$$(x : a?x; x!b) \xrightarrow{a?\tilde{x}} \mathbf{1}; x!b$$

  after which a value for x, presumably generated by communication, can be instantiated later through substitution:

$$spawn(a!c); (x : a?x; x!b) \xrightarrow{\tau} (x : \{x{\leftarrow}c\}; spawn(\mathbf{1}); \mathbf{1}; x!b) \equiv (x : c!b; \{x{\leftarrow}c\}) \equiv c!b$$

  (the equation $spawn(\mathbf{1}) = \mathbf{1}$ is known from the basic calculus).

– A crucial rule is R_{17}, which lifts the transition relation *modulo* structural equivalence. This allows us to "shift" all substitutions out of the way before computing the transitions.

Note that the rules R_8 and R_{16} for sequential composition and communication demand the first operand to fulfil the predicate $\checkmark_\emptyset$. Therefore, in terms $t; u$ with $t\checkmark_\sigma, \sigma \neq \emptyset$, the actions of u can only occur after σ has been applied to u by the rules of structural equivalence. This mechanism prevents non-determinism caused by the application sequence of structural equivalence and transition rules.

Example 3. Consider the process definition $\Theta\colon n(x, y) \mapsto (a : x?a; y!a)$. Note that the variable a, which is bound in the body, must be restricted (otherwise the term would not be well-formed). A process call gives rise to the following behaviour.

$$
\begin{array}{ll}
n(b, c); b!c & \\
\xrightarrow{\tau} & (x, y : \{x \leftarrow b, y \leftarrow c\}; (a : x?a; y!a)); b!c \equiv (x, y, a : b?a; \{x \leftarrow b, y \leftarrow c\}; y!a); b!c \\
\xrightarrow{b?\tilde{a}} & (x, y : \mathbf{1}; \{x \leftarrow b, y \leftarrow c\}; y!a); b!c \qquad \equiv (x, y : c!a; \{x \leftarrow b, y \leftarrow c\}); b!c \\
\xrightarrow{c!a} & (x, y : \mathbf{1}; \{x \leftarrow b, y \leftarrow c\}); b!c \qquad \equiv b!c \\
\xrightarrow{b!c} & \mathbf{1}
\end{array}
$$

Proposition 12. *$\langle \mathcal{L}, \mathcal{F}_{\mathrm{wf}}, \rightarrow, \checkmark \rangle$ is an extended $\checkmark$-transition system.*

Bisimulation. Bisimulation is adapted to the extended termination predicate as follows:

Definition 13. Let T be an extended $\checkmark$-transition system. A symmetrical relation $R \subseteq S \times S$ is called a *bisimulation relation* if for all $(s_1, s_2) \in R$,
– if $s_1 \xrightarrow{\ell} s_1'$ then $\exists s_2 \xrightarrow{\ell} s_2'$ such that $(s_1', s_2') \in R$;
– $s_1\checkmark_\sigma$ then $s_2\checkmark_\sigma$.
$s_1, s_2 \in S$ are called *bisimilar*, denoted $s_1 \mathrel{\dot{\sim}}_{\mathcal{F}} s_2$, if $(s_1, s_2) \in R$ for some bisimulation relation R.

Surprisingly to us, and in contrast to what we claimed in [10], this relation is *not* a congruence over our language. The following example demonstrates this.

Example 4. The terms $t_1 = (a : \mathit{spawn}(a!b); \{x \leftarrow a\})$ and $t_2 = (x : \{x \leftarrow a\})$ are bisimilar: both can terminate with $\{x \leftarrow \hat{a}\}$, and neither can do anything else. In particular, the potential $a!b$-communication of t_1 is prevented by the a-restriction. However, if $u = a?y$ then $t_1; u \nsim t_2; u$: for we can derive

$$
\begin{array}{rl}
t_1; u & \cong (a : \mathit{spawn}(a!b); a?y; \{x \leftarrow a\}) \\
& \xrightarrow{\tau} (a : \{y \leftarrow b\}; \mathit{spawn}(\mathbf{0}); \mathbf{1}; \{x \leftarrow a\}) \\
& \cong (a : \{x \leftarrow a, y \leftarrow b\})
\end{array}
$$

which cannot be matched by $t_2; u$. Here, we see that the $a!b$-communication of t_1 suddenly becomes active in the context $-; u$.

In the next section, we briefly go into possible solutions to this problem. Given the situation, however, we feel that it is not worth the effort to develop the theory for our calculus any further. Note that many of the claims of [10] are hereby invalidated.

4 Conclusions and Future Work

We have presented a direct process algebraic formalisation of operators for process creation and sequential composition, integrated them in a name-passing calculus, and provided this calculus with operational and axiomatic semantics. We now discuss some similar investigations in the literature, possible improvements on the current design of the calculus (in particular with an eye to compositionality w.r.t. bisimulation) and the role of the calculus in the design of the specification language *ProFun*.

Related research. An early formalisation of an operator for process creation is given by Baeten and Vaandrager [3] in the setting of ACP [2, 4], of which sequential composition has always been an integral part. Our basic calculus $\mathcal{B}$ is quite similar to their solution, except that they rely on an auxiliary "asymmetric parallel composition" $\lceil$ such that $t \lceil u$ (in their calculus) precisely corresponds to our $spawn(t); u$. Furthermore, they have a slightly different treatment of termination, due to which they do not need to restrict to guarded choice. However, they do not consider mobility.

Another existing approach along the same lines as ours is the *fork calculus* of Havelund and Larsen [15], extended to the *ψ-calculus* in [14]: they, too, develop a calculus with process creation, sequential composition and name passing. They give a two-level semantics: the first level models the local behaviour of a single process, the second the global system's behaviour as a *multiset* of processes. The latter effectively corresponds to parallel composition restricted to the outermost level; this can again be regarded as an auxiliary operator. The ψ-calculus extension allows name passing in π-calculus style, just as our full calculus $\mathcal{F}$; however, sequential composition is once more restricted to action prefixing, at least for input actions (i.e., the binding constructors). In this way, at the cost of severely restricting the use of sequential composition, the ψ-calculus avoids the problems we have solved by distinguishing between binding and scoping and introducing substitution as a syntactic construct.

The next attempt. As we have seen, the current attempt to formulate a calculus meeting the criteria we had set ourselves had the defect that the natural definition of bisimulation did not give rise to a congruence. We intend to counter this problem by using a notion of (substitution-labelled) termination that is not given by a unary predicate, as in the current calculus, but rather by a transition, as in [3], whose label still carries the substitution that is "announced" to the outside world, but which may induce a simultaneous *scope extrusion* of the names in the range of the substitution. Thus, the behaviour of the terms in the counter-example 4 would become essentially as follows:

$$t_1 \xrightarrow{\{x \leftarrow \hat{a}\}} spawn(a!b) \xrightarrow{a!b} spawn(\mathbf{1}) \qquad t_2 \xrightarrow{\{x \leftarrow \hat{a}\}} \mathbf{1} \not\xrightarrow{a!b}$$

which implies $t_1 \not\sim t_2$.

At the same time, however, we plan to reformulate the semantics in such a way that it no longer relies on structural equivalence; in particular, we want to get rid of Rule R_{17} of Figure 6, since experience has shown us that this greatly complicates proofs conducted by induction on the derivation of transitions —such as congruence proofs.

A semantics for ProFun. The work reported here is part of an ongoing project investigating methods for the design of reactive systems. We are planning to develop a design methodology which allows for a top-down design of systems, based on the

language *ProFun* [9]. The calculus presented in this paper has been developed as a basis for reasoning about the behaviour aspects of *ProFun* programs.

As a next step, in order to reflect all aspects of *ProFun*, we aim to integrate data into the calculus. We are planning to consider names as a representation for functional expressions and identifiers; for instance the function application *f 3 4* will be a valid name and the declaration $x = f\ 3\ 4$ can be translated directly into the substitution $\{x \leftarrow f\ 3\ 4\}$. We claim that this treatment of expressions easily allows for realising *eager* and *lazy evaluation* semantics. Channel values will be represented by a specific data type. The introduction of substitution as a syntactic operator simplifies the operational semantics of the calculus with data, because there is no need for semantic *environments* to reflect bindings (cf. [6]).

Furthermore, the approach of top-down design has to provide mechanisms for the *stepwise* refinement of reactive systems. Therefore, we are planning to adapt techniques for action refinement for our calculus; as mentioned before, this has been another major reason for investigating sequential composition.

Acknowledgements. We are grateful to Heike Wehrheim for comments on this paper. The first author would like to thank Michaela Huhn and Cosimo Laneve for discussions on the integration of concurrency and functional programming.

References

1. J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In E. Best, editor, *Proceedings of CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 1993.
2. J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.
3. Jos C. M. Baeten and Frits W. Vaandrager. An Algebra for Process Creation. *Acta Informatica*, 29(4):303–334, 1992. Report version: CS-R8907, CWI, Amsterdam; also available in "J.W. de Bakker, 25 Jaar Semantiek — Liber Amicorum", Stichting Mathematisch Centrum, Amsterdam, 1989.
4. J.A. Bergstra and J.W. Klop. Algebra for Communicating Processes with Abstraction. *Journal of Theoretical Computer Science*, 37:77–121, 1985.
5. Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation Can't Be Traced. *Journal of the ACM*, 42(1):232–268, January 1995.
6. Gian-Luigi Ferrari, Ugo Montanari, and Paola Quaglia. A π-calculus with explicit substitutions. *Theoretical Computer Science*, 168:53–103, 1996.
7. W. Ferreira and M. Hennessy. Towards a Semantic Theory of CML. Technical Report 2/95, University of Sussex, February 1995.
8. William Ferreira, Matthew Hennessy, and Alan Jeffrey. A Theory of Weak Bisimulation for Core CML. Technical Report 05/95, University of Sussex, September 1995.
9. Thomas Gehrke and Michaela Huhn. ProFun – a Language for Executable Specifications. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP '96)*, volume 1140 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 1996.
10. Thomas Gehrke and Arend Rensink. Process Creation and Full Sequential Composition in a Name-Passing-Calculus. Hildesheimer Informatik-Bericht HIB 7/97, Institut für Informatik, Universität Hildesheim, May 1997.
11. A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2), 1989.
12. Ursula Goltz, Roberto Gorrieri, and Arend Rensink. Comparing Syntactic and Semantic Action Refinement. *Information and Computation*, 125:118–143, 1996.
13. Ursula Goltz and Rob J. van Glabbeek. Equivalence Notions for Concurrent Systems and Refinement of Actions. In *Proceedings of MFCS '89*, Lecture Notes in Computer Science, pages 237–248. Springer, 1989.

14. Klaus Havelund. *The Fork Calculus*. PhD thesis, DIKU, University of Kopenhagen, 1994.
15. Klaus Havelund and Kim G. Larsen. The Fork-Calculus. *Nordic Journal of Computing*, 1:346–363, 1994.
16. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
17. J.F.Groote and F.W.Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, 1992.
18. J.W. Klop. Term Rewriting Systems. In S. Abramsky, D.M. Gabbay, and T.S.E Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford Science Publications, Clarendon Press, 1992.
19. Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
20. Robin Milner. Functions as Processes. Technical Report 1154, INRIA, February 1990.
21. Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I+II. *Information and Computation*, 100, 1992.
22. J.H. Reppy. Concurrent ML: Design, application and semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer, 1992.
23. Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992. No. CST–99–93; also available as ECS–LFCS–93–266.
24. Bent Thomsen, Lone Leth, and Alessandro Giacalone. Some Issues in the Semantics of Facile Distributed Programming. In *Proceedings of REX Workshop "Semantics: Foundations and Applications"*, volume 666 of *Lecture Notes in Computer Science*. Springer, 1992.

A Proofs for the Basic Calculus

Theorem 5. The theory $\mathcal{AX}_{\mathcal{B}}$ is sound with respect to $\sim_{\mathcal{B}}$.

Proof. We prove the soundness of those equations of Figure 2 which are not common in other calculi. The proofs of the remaining equations are omitted. For each axiom of Figure 2 we define a relation and prove that the relation satisfies the bisimulation conditions given in Definition 3.

– Axiom (4): $t;(u;v) = (t;u);v$.
Let R equal the symmetrical closure of $\{(t';(u';v');(t';u');v') \mid t',u',v' \in \mathcal{B}\}$. We prove that it satisfies the bisimulation conditions.
Termination:
  - We assume $t';(u';v')\checkmark$. Then we can deduce with T_3 that $t'\checkmark \wedge u'\checkmark \wedge v'\checkmark$. Therefore, we know with T_3 that $(t';u');v'\checkmark$.
  - For the assumption $(t';u');v'\checkmark$ analogous.

Transitions:
  - We assume $t';(u';v') \xrightarrow{\alpha} t_{new}$ and distinguish between the following cases:
    * $t' \xrightarrow{\alpha} t''$ and $t_{new} = t'';(u';v')$, derived with R_7. Then $(t';u');v' \xrightarrow{\alpha} (t'';u');v'$ with R_7 and $(t'';(u';v'),(t'';u');v') \in R$.
    * $t'\checkmark, u'\checkmark$ and
      · $u' \xrightarrow{\alpha} u''$ such that $t_{new} = t';(u'';v')$, derived with $\mathrm{R}_7, \mathrm{R}_8$. Then $(t';u');v' \xrightarrow{\alpha} (t';u'');v'$ with $\mathrm{R}_8, \mathrm{R}_7$ and $(t';(u'';v'),(t';u'');v') \in R$.
      · $t' \xrightarrow{\gamma} t''$, $u' \xrightarrow{\beta} u''$, $\{\gamma,\beta\} = \{a?,a!\}$, $\alpha = \tau$, $t';u' \xrightarrow{\tau} t'';u''$ and $t_{new} = t'';(u'';v')$, derived with $\mathrm{R}_8, \mathrm{R}_7, \mathrm{R}_9$. Then $(t';u');v' \xrightarrow{\tau} (t'';u'');v'$ with $\mathrm{R}_9, \mathrm{R}_7$ and $(t'';(u'';v'),(t'';u'');v') \in R$.
    * $t'\checkmark, u'\checkmark$ and
      · $v' \xrightarrow{\alpha} v''$ such that $t_{new} = t';(u';v'')$, derived with R_8.
      Then $(t';u');v' \xrightarrow{\alpha} (t';u');v''$ with R_8 and $(t';(u';v''),(t';u');v'') \in R$.

· $t' \xrightarrow{\gamma} t''$, $v' \xrightarrow{\beta} v''$, $\{\gamma, \beta\} = \{a?, a!\}$, $\alpha = \tau$, $t'; v' \xrightarrow{\tau} t''; v''$ and $t_{new} = t''; (u'; v'')$, derived with $\mathrm{R}_8, \mathrm{R}_7, \mathrm{R}_6, \mathrm{R}_9$. Then $(t'; u'); v' \xrightarrow{\tau} (t''; u'); v''$ with $\mathrm{R}_8, \mathrm{R}_7, \mathrm{R}_6, \mathrm{R}_9$ and $(t''; (u'; v''), (t''; u'); v'') \in R$.

· $u' \xrightarrow{\gamma} u''$, $v' \xrightarrow{\beta} v''$, $\{\gamma, \beta\} = \{a!, a?\}$, $\alpha = \tau$, $u'; v' \xrightarrow{\tau} u''; v''$ and $t_{new} = t'; (u''; v'')$, derived with $\mathrm{R}_9, \mathrm{R}_8$. Then $(t'; u'); v' \xrightarrow{\tau} (t'; u''); v''$ with $\mathrm{R}_7, \mathrm{R}_8, \mathrm{R}_9$ and $(t'; (u''; v''), (t'; u''); v'') \in R$.

• For the assumption $(t'; u'); v' \xrightarrow{\alpha} t'_{new}$ analogous.

– Axiom (9): $(t + u); v = t; v + u; v$.
Let R equal the symmetrical closure of

$$\{((t' + u'); v', t'; v' + u'; v') \mid t', u', v' \in \mathcal{B}\} \cup \{(t', t') \mid t' \in \mathcal{B}\} \ .$$

We prove that it satisfies the bisimulation conditions.
Termination: Due to the restriction to guarded choice we have $(t'; v' + u'; v')\checkmark$ and $(t' + u')\checkmark$, which implies $(t' + u'); v'\checkmark$ with T_3.
Transitions:

• We assume $(t' + u'); v' \xrightarrow{\alpha} t_{new}$ and distinguish the following cases:
1. $t' \xrightarrow{\alpha} t''$ such that $t_{new} = t''; v'$, derived with R_3,R_7. Then $t'; v' + u'; v' \xrightarrow{\alpha} t''; v'$ with R_3,R_7 and $(t''; v', t''; v') \in R$.
2. $u' \xrightarrow{\alpha} u''$ such that $t_{new} = u''; v'$, derived with R_4,R_7. Then $t'; v' + u'; v' \xrightarrow{\alpha} u''; v'$ and $(u''; v', u''; v') \in R$.

• For the assumption $t'; v' + u'; v' \xrightarrow{\alpha} t'_{new}$ analogous.

– Axiom (18): $(a : spawn(t)) = spawn((a : t))$.
Let R equal the symmetrical closure of

$$\{((a : spawn(u)), spawn((a : u))) \mid a \in \mathcal{C}, u \in \mathcal{B}\}.$$

We prove that it satisfies the bisimulation conditions.
Termination: With T_2 we know that $\forall v \in \mathcal{B} : spawn(v)\checkmark$. Then we can deduce $spawn(u)\checkmark$ and $spawn((a : u))\checkmark$. With $spawn(u)\checkmark$ and T_4 we can deduce $(a : spawn(u))\checkmark$. Therefore, both sides of the pairs in R are terminated.
Transitions:

• Assume $(a : spawn(u)) \xrightarrow{\alpha} t_{new}$. The only possible transition is $u \xrightarrow{\alpha} u'$ and we can derive $t_{new} = (a : spawn(u')) \wedge \alpha \notin \{a?, a!\}$ with R_6 and R_5. Therefore, $spawn((a : u)) \xrightarrow{\alpha} spawn((a : u'))$ with R_5 and R_6. Furthermore, $((a : spawn(u')), spawn((a : u'))) \in R$.

• Assume $spawn((a : u)) \xrightarrow{\alpha} t'_{new}$. Then $u \xrightarrow{\alpha} u'$ and we can derive $t'_{new} = spawn((a : u')) \wedge \alpha \notin \{a?, a!\}$ with R_5 and R_6. Therefore, $(a : spawn(u)) \xrightarrow{\alpha} (a : spawn(u'))$ with R_6 and R_5. Furthermore, $((a : spawn(u')), spawn((a : u'))) \in R$.

– Axiom (19): $spawn(\mathbf{0}) = \mathbf{1}$.
Let R be the symmetrical closure of $\{(spawn(\mathbf{0}), \mathbf{1})\}$. We prove that it satisfies the bisimulation conditions.
Termination: With T_1 we know that $\mathbf{1}\checkmark$. With T_2 we know that $\forall t \in \mathcal{B} : spawn(t)\checkmark$. Therefore, $spawn(\mathbf{0})\checkmark$. Both sides of the pairs in R are terminated.
Transitions: With $\mathbf{0} \not\rightarrow$ we can deduce $spawn(\mathbf{0}) \not\rightarrow$. Similarly $\mathbf{1} \not\rightarrow$. Therefore, neither side of the pairs in R can perform a transition.

– Axiom (20): $spawn(t); spawn(u) = spawn(u); spawn(t)$.
Let R be

$$\{(spawn(t'); spawn(u'), spawn(u'); spawn(t')) \mid t', u' \in \mathcal{B}\}.$$

We prove that it satisfies the bisimulation conditions.
Termination: With T_2 we know that $\forall v \in \mathcal{B} : spawn(v)\checkmark$. Therefore, $spawn(t')\checkmark$ and $spawn(u')\checkmark$. Furthermore, we know with T_3 that $spawn(t'); spawn(u')\checkmark$ and $spawn(u'); spawn(t')\checkmark$.
Transitions:

- We assume $spawn(t'); spawn(u') \xrightarrow{\alpha} t_{new}$ and distinguish between the following cases:
  - $t' \xrightarrow{\alpha} t''$ and $t_{new} = spawn(t''); spawn(u')$, derived with R_7,R_6. Then $spawn(u'); spawn(t') \xrightarrow{\alpha} spawn(u'); spawn(t'')$, derived with R_8,R_6. Furthermore, $(spawn(t''); spawn(u'), spawn(u'); spawn(t'')) \in R$.
  - $u' \xrightarrow{\alpha} u''$ and $t_{new} = spawn(t'); spawn(u'')$, derived with R_8,R_6. Then $spawn(u'); spawn(t') \xrightarrow{\alpha} spawn(u''); spawn(t')$, derived with R_7,R_6. Furthermore, $(spawn(t'); spawn(u''), spawn(u''); spawn(t')) \in R$.
  - $t' \xrightarrow{\gamma} t''$, $u' \xrightarrow{\beta} u''$, $\{\gamma, \beta\} = \{a?, a!\}$, $\alpha = \tau$ and $t_{new} = spawn(t''); spawn(u'')$, derived with R_7, R_8, R_9 and R_6. Then $spawn(u'); spawn(t') \xrightarrow{\tau} spawn(u''); spawn(t'')$, derived with R_7, R_8, R_9 and R_6. Furthermore, $(spawn(t''); spawn(u''), spawn(u''); spawn(t'')) \in R$.
- For the assumption $spawn(u'); spawn(t') \xrightarrow{\alpha} t'_{new}$ analogous.

– Axiom (21): $spawn(t); spawn(u) = spawn(spawn(t); u)$.
Let R equal the symmetrical closure of $\{(spawn(t'); spawn(u'), spawn(spawn(t'); u')) \mid t', u' \in \mathcal{B}\}$. We prove that it satisfies the bisimulation conditions.
Termination: With T_2 we know that $\forall v \in \mathcal{B} : spawn(v)\checkmark$. Therefore, $spawn(t')\checkmark$, $spawn(u')\checkmark$ and $spawn(spawn(t'); u')\checkmark$. With $spawn(t')\checkmark$, $spawn(u')\checkmark$ and T_3 we can deduce that $spawn(t); spawn(u)\checkmark$. Therefore, both sides of the pairs in R are terminated.
Transitions:

- We assume $spawn(t'); spawn(u') \xrightarrow{\alpha} t_{new}$ and distinguish between the following cases:
  - $t' \xrightarrow{\alpha} t''$ and $t_{new} = spawn(t''); spawn(u')$, derived with R_7 and R_6. Therefore, $spawn(spawn(t'); u') \xrightarrow{\alpha} spawn(spawn(t''); u')$ with R_6 and R_7. Furthermore, $(spawn(t''); spawn(u'), spawn(spawn(t''); u')) \in R$.
  - $u' \xrightarrow{\alpha} u''$ and $t_{new} = spawn(t'); spawn(u'')$, derived with R_8 and R_6. Therefore, $spawn(spawn(t'); u') \xrightarrow{\alpha} spawn(spawn(t'); u'')$ with R_6 and R_8. Furthermore, $(spawn(t'); spawn(u''), spawn(spawn(t'); u'')) \in R$.
  - $t' \xrightarrow{\gamma} t''$, $u' \xrightarrow{\beta} u''$, $\{\gamma, \beta\} = \{a?, a!\}$, $\alpha = \tau$, $t_{new} = spawn(t''); spawn(u'')$ with R_6. Then $spawn(spawn(t'); u') \xrightarrow{\tau} spawn(spawn(t''); u'')$ with R_9 and R_6. Furthermore, $(spawn(t''); spawn(u''), spawn(spawn(t''); u'')) \in R$.
- For the assumption $spawn(spawn(t'); u') \xrightarrow{\alpha} t'_{new}$ analogous.

– Axiom (26): $spawn(t; spawn(u)) = spawn(t; u)$.
Let R equal the symmetrical closure of

$$\{(spawn(t'; spawn(u')), spawn(t'; u')) \mid t', u' \in \mathcal{B}\}.$$

We prove that it satisfies the bisimulation conditions.
Termination: With T_2 we know that $\forall v \in \mathcal{B} : spawn(v)\checkmark$. Therefore, $spawn(t'; spawn(u'))\checkmark$ and $spawn(t'; u')\checkmark$.
Transitions:

- We assume $spawn(t'; spawn(u')) \xrightarrow{\alpha} t_{new}$ and distinguish between the following cases:
  - $t' \xrightarrow{\alpha} t''$ and $t_{new} = spawn(t''; spawn(u'))$, derived with R_7, R_6. Then $spawn(t'; u') \xrightarrow{\alpha} spawn(t'', u')$ with R_7, R_6. Furthermore, $(spawn(t''; spawn(u')), spawn(t''; u')) \in R$.
  - $u' \xrightarrow{\alpha} u''$ and $t_{new} = spawn(t'; spawn(u''))$ and $t'\checkmark$, derived with R_6, R_8. Then $spawn(t'; u') \xrightarrow{\alpha} spawn(t; u'')$ with R_6, R_8. Furthermore, $(spawn(t'; spawn(u'')), spawn(t'; u'')) \in R$.
  - $t'\checkmark$, $t' \xrightarrow{\gamma} t''$, $u' \xrightarrow{\beta} u''$, $\{\gamma, \beta\} = \{a?, a!\}$ and $\alpha = \tau$, such that $t_{new} = spawn(t''; spawn(u''))$ with R_6. Then $spawn(t'; u') \xrightarrow{\tau} spawn(t''; u'')$ with R_6.
    Furthermore, $(spawn(t''; spawn(u'')), spawn(t''; u'')) \in R$.
- For the assumption $spawn(t'; u') \xrightarrow{\alpha} t'_{new}$ analogous.

– Axiom (22): $spawn(t; spawn(u) + v) = spawn(t; u + v)$.
Let $R_1 = \{(spawn(t'; spawn(u') + v'), spawn(t'; u' + v')) \mid t', u', v' \in \mathcal{B}\}$, let $R_2 = \{(t', t') \mid t' \in \mathcal{B}\}$ and let $R_3 = \{(spawn(t'; spawn(u')), spawn(t', u')) \mid t', u' \in \mathcal{B}\}$. Let R equal the symmetrical closure of $R_1 \cup R_2 \cup R_3$. We prove that it satisfies the bisimulation conditions.
Termination: With T_2 we know that $\forall v \in \mathcal{B} : spawn(v)\checkmark$. Therefore, $spawn(t'; spawn(u') + v')\checkmark$ and $spawn(t'; u' + v')\checkmark$. R_2 is trivial; for the part R_3 see the previous axiom (26).
Transitions:
- We assume $spawn(t'; spawn(u') + v') \xrightarrow{\alpha} t_{new}$ and distinguish between the following cases:
  - $v' \xrightarrow{\alpha} v''$ and $t_{new} = spawn(v'')$, derived with R_4, R_6. Then $spawn(t'; u' + v') \xrightarrow{\alpha} spawn(v'')$ with R_4, R_6 and $(spawn(v''), spawn(v'')) \in R$.
  - $t' \xrightarrow{\alpha} t''$. Then $t_{new} = spawn(t''; spawn(u'))$, derived with R_3, R_6. Then $spawn(t'; u' + v') \xrightarrow{\alpha} spawn(t''; u')$ with R_3, R_6 and $(spawn(t''; spawn(u')), spawn(t'', u')) \in R$.
- For the assumption $spawn(t'; u' + v') \xrightarrow{\alpha} t'_{new}$ analogous.
- R_2 is trivial.
- For R_3 see the previous axiom (26).

– Axiom (23): *Expansion law.*
Let $t' = \sum_{i \in \mathcal{I}} \alpha_i; t'_i$, let $u' = \sum_{k \in \mathcal{K}} \beta_k; u'_k$. Let

$$R_1 = \left\{ \begin{array}{l} (spawn(t'); u', \\ \sum_{i \in \mathcal{I}} \alpha_i; spawn(t'_i); u' + \sum_{k \in \mathcal{K}} \beta_k; spawn(t'); u'_k \\ + \sum_{\{\alpha_i, \beta_k\} = \{a?, a!\}} \tau; spawn(t'_i); u'_k) \mid t', u', t'_i, u'_k \in \mathcal{B} \end{array} \right\},$$
$$R_2 = \{(spawn(\mathbf{1}; t'_i); u', \mathbf{1}; spawn(t'_i); u') \mid t'_i, u' \in \mathcal{B}\},$$
$$R_3 = \{(spawn(t'); \mathbf{1}; u'_k, \mathbf{1}; spawn(t'); u'_k) \mid t', u'_k \in \mathcal{B}\},$$
$$R_4 = \{(spawn(\mathbf{1}; t'_i); \mathbf{1}; u'_k, \mathbf{1}; spawn(t'_i); u'_k) \mid t'_i, u'_k \in \mathcal{B}\}.$$

Let R be the symmetrical closure of $R_1 \cup R_2 \cup R_3 \cup R_4$. We prove that it satisfies the bisimulation conditions. We denote the left hand elements of the pairs in R by t_L and the right hand elements by t_R.
Termination: For R_1 we know that u' is guarded, therefore $u'\checkmark$. With T_3 we know that $t_L\checkmark$. Furthermore, t_R is guarded, therefore $t_R\checkmark$. Both sides of the pairs in R do not terminate. For R_2, R_3 and R_4 we can remove the $\mathbf{1}$-operators with the axioms (1), (2) using congruence. Then both elements of each pair are

identical and therefore bisimilar. This implies that they have the same termination behaviour.

Transitions:

- Regarding R_1, we assume $t_L \xrightarrow{\alpha} t_{new}$ and distinguish the between following cases:
  - $\exists i \in \mathcal{I} : \alpha_i \xrightarrow{\alpha_i} \mathbf{1}$ and $t_{new} = spawn(\mathbf{1}; t_i'); u'$, derived with $\mathrm{R}_7, \mathrm{R}_6$ and the choice rules. Then $t_R \xrightarrow{\alpha_i} \mathbf{1}; spawn(t_i'); u'$ with R_7 and the choice rules. Furthermore, $(spawn(\mathbf{1}; t_i'); u', \mathbf{1}; spawn(t_i'); u') \in R$.
  - $\exists k \in \mathcal{K} : \beta_k \xrightarrow{\beta_k} \mathbf{1}$ and $t_{new} = spawn(t'); \mathbf{1}; u_k'$, derived with $\mathrm{R}_8, \mathrm{R}_7$ and the choice rules. Then $t_R \xrightarrow{\beta_k} \mathbf{1}; spawn(t'); u_k'$ with R_7 and the choice rules. Furthermore, $(spawn(t'); \mathbf{1}; u_k', \mathbf{1}; spawn(t'); u_k') \in R$.
  - $\exists i \in \mathcal{I} \exists k \in \mathcal{K} \exists a \in \mathcal{C} : \alpha_i \xrightarrow{\alpha_i} \mathbf{1} \wedge \beta_k \xrightarrow{\beta_k} \mathbf{1} \wedge \{\alpha_i, \beta_k\} = \{a?, a!\}$, $\alpha = \tau$ and $t_{new} = spawn(\mathbf{1}; t_i'); \mathbf{1}; u_k'$, derived with R_9. Then $t_R \xrightarrow{\tau} \mathbf{1}; spawn(t_i'); u_k'$ with R_4. Furthermore, $(spawn(\mathbf{1}; t_i'); \mathbf{1}; u_k', \mathbf{1}; spawn(t_i'); u_k') \in R$.
- We assume $t_R \xrightarrow{\alpha} t_{new}'$ and distinguish between the following cases:
  - $\exists i \in \mathcal{I} : \alpha_i \xrightarrow{\alpha_i} \mathbf{1}$ and $t_{new}' = \mathbf{1}; spawn(t_i'); u'$ with R_7 and the choice rules. Then $t_L \xrightarrow{\alpha_i} spawn(\mathbf{1}; t_i'); u'$ with $\mathrm{R}_7, \mathrm{R}_6$ and the choice rules. Furthermore, $(spawn(\mathbf{1}; t_i'); u', \mathbf{1}; spawn(t_i'); u') \in R$.
  - $\exists k \in \mathcal{K} : \beta_k \xrightarrow{\beta_k} \mathbf{1}$ and $t_{new}' = \mathbf{1}; spawn(t'); u_k'$ with R_7 and the choice rules. Then $t_L \xrightarrow{\beta_k} spawn(t'); \mathbf{1}; u_k'$ with $\mathrm{R}_8, \mathrm{R}_7$ and the choice rules. Furthermore, $(spawn(t'); \mathbf{1}; u_k', \mathbf{1}; spawn(t'); u_k') \in R$.
  - $\exists i \in \mathcal{I} \exists k \in \mathcal{K} \exists a \in \mathcal{C} : \alpha_i \xrightarrow{\alpha_i} \mathbf{1} \wedge \beta_k \xrightarrow{\beta_k} \mathbf{1} \wedge \{\alpha_i, \beta_k\} = \{a?, a!\}$, $\alpha = \tau$ and $t_{new}' = \mathbf{1}; spawn(t_i'); u_k'$, derived with R_7 and the choice rules. Then $t_L \xrightarrow{\tau} spawn(\mathbf{1}; t_i'); \mathbf{1}; u_k$ with $\mathrm{R}_9, \mathrm{R}_6$ and the choice rules. Furthermore, $(spawn(\mathbf{1}; t_i'); \mathbf{1}; u_k', \mathbf{1}; spawn(t_i'); u_k') \in R$.
- For the transitions of R_2, R_3, R_4 see axioms (1) and (2) combined with congruence.

□

Theorem 7. For all $t \in \mathcal{B}_{fin}$, there is an $u \in \mathcal{B}_{fin}$ in basic normal form such that $\mathcal{AX}_{\mathcal{B}} \vdash t = u$.

We define two functions $sbnf : \mathcal{B}_{fin} \rightharpoonup \mathcal{B}_{fin}$, $bnf : \mathcal{B}_{fin} \to \mathcal{B}_{fin}$. $bnf(t)$ computes the basic normal form of t. Given a term t in basic normal form, $sbnf(t)$ generates the simple basic normal of t. Therefore, $t \sim_{\mathcal{B}} sbnf(t)$ is not valid for all terms t in bnf, but we can deduce that $spawn(t) \sim_{\mathcal{B}} spawn(sbnf(t))$.

First, we define the function $sbnf$. It is applied to terms t in basic normal form and removes all the *spawn*-operations in t.

$$sbnf(t) = \begin{cases} \sum \{\} & \text{if } t = \sum \{\} \\ sbnf(t') & \text{if } t = spawn(t') \\ \sum_{i \in \mathcal{I}} \alpha_i; sbnf(t_i) & \text{if } t = \sum_{i \in \mathcal{I}} \alpha_i; t_i \end{cases}$$

We have to prove that the function $sbnf$ always terminates.

Lemma 14. $\forall t \in \mathcal{B}_{fin}, t \text{ in } bnf, \exists u \in \mathcal{B}_{fin} : sbnf(t) = u$.

Proof. We define the size of a term, which represents the number of its operators. Let $size : \mathcal{B}_{fin} \to \mathbf{N}$ be a function, which is defined as follows:

$$size(t) = \begin{cases} 1 & \text{if } t = \mathbf{0} \text{ or } t = \mathbf{1} \text{ or } t = a? \text{ or } t = a! \\ size(t') + 1 & \text{if } t = spawn(t') \text{ or } t = (a : t') \\ size(t_1) + size(t_2) + 1 & \text{if } t = t_1 + t_2 \text{ or } t = t_1; t_2 \end{cases}$$

It is easy to verify that in each defining equation for *sbnf* the arguments of the function on the left hand side have a greater size than the arguments of the right hand side. This means that the size of the terms to which *sbnf* is applied is decreased in each application step.

Furthermore, we have to show that *sbnf* generates terms in simple basic normal form (sbnf) for terms in bnf.

Lemma 15. $\forall t \in \mathcal{B}_{fin}, t$ *in bnf:* $sbnf(t)$ *is in sbnf.*

Proof. We prove the lemma via induction over the definition structure of *sbnf*.

- $\sum \{\}$ is in sbnf.
- $sbnf(t)$ is in *sbnf* by induction hypothesis.
- $\sum_{i \in \mathcal{I}} \alpha_i; sbnf(t_i)$ is in *sbnf*, because $\forall i \in \mathcal{I} : sbnf(t_i)$ is in *sbnf* by induction hypothesis.

Finally, we have to prove that $\forall t \in \mathcal{B}_{fin}, t$ in $bnf : spawn(t) \sim_{\mathcal{B}} spawn(sbnf(t))$.

Lemma 16. $\forall t \in \mathcal{B}_{fin}, t$ *in bnf* : $spawn(t) \sim_{\mathcal{B}} spawn(sbnf(t))$.

Proof. We prove the lemma via induction over the size of term t.

- $size(t) = 1$. Then $t = \sum \{\}$ and $sbnf(t) = \sum \{\}$. $spawn(\sum \{\}) = spawn(\sum \{\})$ is trivial.
- $size(t) = n, n > 1$. We assume as induction hypothesis that for all v in bnf with $size(v) < n$: $spawn(v) = spawn(sbnf(v))$.
  - $t = spawn(t')$. With axiom (25) we have $spawn(spawn(t')) = spawn(t') = spawn(sbnf(t'))$.
  - $t = \sum_{i \in \mathcal{I}} \alpha_i; t_i$.

$$
\begin{array}{ll}
& spawn(\sum_{i \in \mathcal{I}} \alpha_i; t_i) \\
= & spawn(\sum_{i \in \mathcal{I}} \alpha_i; spawn(t_i)) \quad \text{(axiom (22))} \\
= & spawn(\sum_{i \in \mathcal{I}} \alpha_i; spawn(sbnf(t_i))) \quad \text{(induction hypothesis)} \\
= & spawn(\sum_{i \in \mathcal{I}} \alpha_i; sbnf(t_i)) \quad \text{(axiom (22))}
\end{array}
$$

With *sbnf* we can now define the function $bnf : \mathcal{B}_{fin} \to \mathcal{B}_{fin}$, which generates terms in basic normal form (bnf). Let t, u be in bnf.

$$
\begin{array}{l}
bnf(\mathbf{0}) = \sum \{\} \\
bnf(\mathbf{1}) = spawn(\sum \{\}) \\
bnf(a!) = a!; spawn(\sum \{\}) \\
bnf(a?) = a?; spawn(\sum \{\}) \\
bnf(spawn(t)) = spawn(sbnf(t)) \\
bnf(t + u) = t + u \\
bnf((a : t)) = \begin{cases} \sum\limits_{\substack{i \in \mathcal{I} \\ \alpha_i \notin \{a?, a!\}}} \alpha_i; bnf((a : t_i)) & \text{if } t = \sum_{i \in \mathcal{I}} \alpha_i; t_i \\ spawn(sbnf(bnf((a : t')))) & \text{if } t = spawn(t') \end{cases}
\end{array}
$$

$$bnf(t;u) = \begin{cases} \sum_{i\in\mathcal{I}} \alpha_i; bnf(t_i;u) \\ \quad \text{if } t = \sum_{i\in\mathcal{I}} \alpha_i; t_i \\ \sum_{i\in\mathcal{I}} \alpha_i; bnf(spawn(t_i);u) \\ + \sum_{k\in\mathcal{K}} \beta_k; bnf(spawn(t);u_k) \\ + \sum_{\{\alpha_i,\beta_k\}=\{a?,a!\}} \tau; bnf(spawn(t_i);u_k) \\ \quad \text{if } t = spawn(\sum_{i\in\mathcal{I}} \alpha_i; t_i), u = \sum_{k\in\mathcal{K}} \beta_k; u_k \\ spawn(sbnf(bnf(spawn(t');u'))) \\ \quad \text{if } t = spawn(t') \text{ and } u = spawn(u') \end{cases}$$

Analogous to $sbnf$, we have to prove that the function bnf always terminates.

Lemma 17. $\forall t \in \mathcal{B}_{fin} \exists u \in \mathcal{B}_{fin} : bnf(t) = u$.

Proof. We use the function *size* from Lemma 14. It is easy to verify that in each defining equation for bnf the arguments of bnf on the left hand side have a greater size than the arguments of the right hand side. This means, that the size of the terms to which bnf is applied is decreased in each application step. Therefore, the function bnf terminates for every term $t \in \mathcal{B}_{fin}$.

Furthermore, we have to show, that bnf generates terms in basic normal form.

Lemma 18. $\forall t \in \mathcal{B}_{fin} : bnf(t)$ *is in bnf.*

Proof. We prove the lemma via induction over the size of the terms.

- $\sum\{\}$ is in bnf, $spawn(\mathbf{0})$ is in bnf, $a!; spawn(\mathbf{0})$ is in bnf, $a?; spawn(\mathbf{0})$ is in bnf.
- $spawn(sbnf(t))$ is in bnf, because t is in bnf by induction hypothesis and $sbnf(t)$ is in simple basic normal form (Lemma 15).
- $t + u$ is in bnf, because t and u are in bnf by induction hypothesis and the summation of two bnfs is still in bnf.
- for $bnf((a:t))$:
  - $\sum_{\substack{i\in\mathcal{I} \\ \alpha_i\notin\{a?,a!\}}} \alpha_i; bnf((a:t_i))$ is in bnf, because for all $i \in \mathcal{I} : bnf((a:t_i))$ is in bnf by induction hypothesis. Then for all $i \in \mathcal{I} : \alpha_i; bnf((a:t_i))$ is in bnf. Furthermore, the summation of bnfs is in bnf.
  - $spawn(sbnf(bnf((a:t'))))$ is in bnf, because we know with the condition $t = spawn(t')$ that t' must be in sbnf. Then $sbnf(bnf((a:t')))$ must be in sbnf and therefore $spawn(bnf((a:t')))$ is in bnf.
- for $bnf(t;u)$:
  - $\sum_{i\in\mathcal{I}} \alpha_i; bnf(t_i;u)$ is in bnf, because $\forall i \in \mathcal{I} : bnf(t_i;u)$ is in bnf by induction hypothesis and the summation of bnfs is a bnf.
  - $\sum_{i\in\mathcal{I}} \alpha_i; bnf(spawn(t_i);u) + \sum_{k\in\mathcal{K}} \beta_k; bnf(spawn(t);u_k) + \sum_{\{\alpha_i,\beta_k\}=\{a?,a!\}} \tau; bnf(spawn(t_i);u_k)$ is in bnf.
    $bnf(spawn(t_i);u)$, $bnf(spawn(t);u_k)$, $bnf(spawn(t_i);u_k)$ are in bnf by induction hypothesis. α_i is a single action, therefore $\alpha_i; bnf(spawn(t_i);u)$ is in bnf (analogous for β_k, τ). The summation of bnfs is also in bnf.
  - $spawn(sbnf(bnf(spawn(t');u')))$ is in bnf, because $bnf(spawn(t');u')$ is in bnf by induction hypothesis and $sbnf(spawn(t');u')$ is in sbnf (Lemma 15).

Finally, we have to prove that $\forall t \in \mathcal{B}_{fin} : t \sim_{\mathcal{B}} bnf(t)t$.

Lemma 19. $\forall t \in \mathcal{B}_{fin} : \mathcal{AX}_{\mathcal{B}} \vdash t = bnf(t)$.

Proof. $-\sum\{\} = \mathbf{0}$ per definition.
- $\mathbf{1} = \mathit{spawn}(\mathbf{0})$ (axiom (19)) $= \mathit{spawn}(\sum\{\})$ (previous case).
- $a\dagger = a\dagger;\mathbf{1}$ (axiom (2)) $= a\dagger;\mathit{spawn}(\mathbf{0})$ (axiom (19)) $= a\dagger;\mathit{spawn}(\sum\{\})$ (first case).
- $\mathit{spawn}(t) = \mathit{spawn}(\mathit{sbnf}(t))$ with t in bnf by induction hypothesis and Lemma 16.
- $t + u = \mathit{bnf}(t) + \mathit{bnf}(u)$, because $t = \mathit{bnf}(t)$ and $u = \mathit{bnf}(u)$ by induction hypothesis and $\sim_{\mathcal{B}}$ is a congruence.
- $t = (a : t')$. We distinguish between the following cases:
  - $\mathit{bnf}(t') = \sum_{i\in\mathcal{I}} \alpha_i; t_i$. Then $(a : t') = (a : \sum_{i\in\mathcal{I}} \alpha_i; t_i)$ (induction hypothesis) $= \sum_{i\in\mathcal{I}}(a : \alpha_i; t_i)$ (axiom (17)) $= \sum_{\substack{i\in\mathcal{I} \\ \alpha_i\notin\{a?,a!\}}} \alpha_i; (a : t_i)$ (axioms (15), (16), (5)) $= \sum_{\substack{i\in\mathcal{I} \\ \alpha_i\notin\{a?,a!\}}} \alpha_i; \mathit{bnf}((a : t_i))$ (induction hypothesis and congruence of $\sim_{\mathcal{B}}$ for sequential composition and choice).
  - $\mathit{bnf}(t') = \mathit{spawn}(t_1)$. Then $(a : t') = (a : \mathit{spawn}(t_1))$ (induction hypothesis) $= \mathit{spawn}((a : t_1))$ (axiom (18)) $= \mathit{spawn}(\mathit{sbnf}(\mathit{bnf}((a : t_1))))$ (induction hypothesis and Lemma 16).
- $t = t'; u$. We distinguish between the following cases:
  - $\mathit{bnf}(t') = \sum_{i\in\mathcal{I}} \alpha_i; t_i$. Then $t = (\sum_{i\in\mathcal{I}} \alpha_i; t_i); u$ (induction hypothesis and $\sim_{\mathcal{B}}$ is a congruence for sequential composition) $= \sum_{i\in\mathcal{I}} \alpha_i; (t_i; u)$ (axiom (9)). By induction hypothesis we have $\forall i \in \mathcal{I} : t_i; u = \mathit{bnf}(t_i; u)$. With the congruence property of $\sim_{\mathcal{B}}$ we can deduce $t = \sum_{i\in\mathcal{I}} \alpha_i; \mathit{bnf}(t_i; u)$.
  - $\mathit{bnf}(t') = \mathit{spawn}(\sum_{i\in\mathcal{I}} \alpha_i; t_i)$, $\mathit{bnf}(u) = \sum_{k\in\mathcal{K}} \beta_k; u_k$.
    Then $t'; u = \mathit{spawn}(\sum_{i\in\mathcal{I}} \alpha_i; t_i); \sum_{k\in\mathcal{K}} \beta_k; u_k$, derived with the induction hypothesis and the congruence property of $\sim_{\mathcal{B}}$ for sequential composition. Then $t'; u = \sum_{i\in\mathcal{I}} \alpha_i; \mathit{spawn}(t_i); u + \sum_{k\in\mathcal{K}} \beta_k; \mathit{spawn}(t); u_k +$
    $\sum_{\{\alpha_i,\beta_k\}=\{a?,a!\}} \tau; \mathit{spawn}(t_i); u_k$ with axiom (23). By induction hypothesis we have $\forall i \in \mathcal{I} : \mathit{spawn}(t_i); u = \mathit{bnf}(\mathit{spawn}(t_i); u)$, $\forall k \in \mathcal{K} : \mathit{spawn}(t); u_k = \mathit{bnf}(\mathit{spawn}(t); u_k)$ and $\forall i \in \mathcal{I} \forall k \in \mathcal{K} : \mathit{spawn}(t_i); u_k = \mathit{bnf}(\mathit{spawn}(t_i); u_k)$. With the congruence property of $\sim_{\mathcal{B}}$ for sequential composition and choice we can deduce $t = \sum_{i\in\mathcal{I}} \alpha_i; \mathit{bnf}(\mathit{spawn}(t_i); u)$
    $+ \sum_{k\in\mathcal{K}} \beta_k; \mathit{bnf}(\mathit{spawn}(t); u_k) + \sum_{\{\alpha_i,\beta_k\}=\{a?,a!\}} \tau; \mathit{bnf}(\mathit{spawn}(t_i); u_k)$.
  - $\mathit{bnf}(t') = \mathit{spawn}(t_1)$, $\mathit{bnf}(u) = \mathit{spawn}(u_1)$. Then $t'; u = \mathit{spawn}(t_1); \mathit{spawn}(u_1)$, derived by induction hypothesis and the congruence property of $\sim_{\mathcal{B}}$ for sequential composition. Then $\mathit{spawn}(t_1); \mathit{spawn}(u_1) = \mathit{spawn}(\mathit{spawn}(t_1); u_1)$ with axiom (21). Finally, we can deduce with Lemma 16 and the induction hypothesis that $\mathit{spawn}(\mathit{spawn}(t_1); u_1) = \mathit{spawn}(\mathit{sbnf}(\mathit{bnf}(\mathit{spawn}(t_1); u_1)))$.

□

Proof of Theorem 7. By the combination of Lemma 17, Lemma 18 and Lemma 19 we have proved the theorem.

□

Theorem 8: The theory $\mathcal{AX}_{\mathcal{B}}$ is complete for $\sim_{\mathcal{B}}$ on the finite fragment $\mathcal{B}_{\mathit{fin}}$ of $\mathcal{B}$.

We have to prove that $\forall t; u \in \mathcal{B}_{\mathit{fin}} : t \sim_{\mathcal{B}} u \Rightarrow \mathcal{AX}_{\mathcal{B}} \vdash t = u$. We extend the proof system for $\mathcal{AX}_{\mathcal{B}} \vdash \_ = \_$ by the commonly used proof rules for the congruence of $\sim_{\mathcal{B}}$ w.r.t. the operators in $\mathcal{B}$.

We define a function $\mathit{depth} : \mathcal{B}_{\mathit{fin}} \rightharpoonup \mathbf{N}$, which computes the depth of a given term in basic normal form (bnf):

$$depth(t) = \begin{cases} 0 & \text{if } t = spawn(t') \\ 0 & \text{if } t = \sum\{\} \\ max_{i\in\mathcal{I}}\{1 + depth(t_i)\} & \text{if } t = \sum_{i\in\mathcal{I}} \alpha_i; t_i, \mathcal{I} \neq \emptyset \end{cases}$$

First, we prove the theorem for *simple basic normal forms*.

Lemma 20. *$\forall t, u \in \mathcal{B}_{fin}, t, u$ in simple basic normal form: $t \sim_{\mathcal{B}} u \Rightarrow \mathcal{AX}_{\mathcal{B}} \vdash t = u$.*

Proof. Let $t, u \in \mathcal{B}_{fin}$ in simple bnf with $t \sim_{\mathcal{B}} u$. We prove the theorem by induction over $d = depth(t) + depth(u)$.

- $d = 0$. Then $t = \sum\{\}$ and $u = \sum\{\}$. Therefore, the result follows by the reflexity rule.
- $d > 0$. Then $t = \sum_{i\in\mathcal{I}} \alpha_i; t_i$ and $u = \sum_{k\in\mathcal{K}} \beta_k; u_k$. For all $i \in \mathcal{I} : t \xrightarrow{\alpha_i} \mathbf{1}; t_i$. With $t \sim_{\mathcal{B}} u$ we know that $\exists k_i \in \mathcal{K} : u \xrightarrow{\beta_{k_i}} \mathbf{1}; u_{k_i}$ with $\alpha_i = \beta_{k_i}$ and $t_i \sim_{\mathcal{B}} u_{k_i}$. By the induction hypothesis we can deduce that $\mathcal{AX}_{\mathcal{B}} \vdash \mathbf{1}; t_i = \mathbf{1}; u_{k_i}$ and therefore $\mathcal{AX}_{\mathcal{B}} \vdash t_i = u_{k_i}$. $\sim_{\mathcal{B}}$ is a congruence for sequential composition, therefore we can apply the congruence rule for ; to get $\mathcal{AX}_{\mathcal{B}} \vdash \alpha_i; t_i = \alpha_i; u_{k_i}$. Similarly, each summand of u can be proved to be equal to a summand of t. Axiom (7) can be used for the removal of duplicate summands. The axioms (6) and (8) can be used to reorder and regroup summands if necessary. $\sim_{\mathcal{B}}$ is a congruence for $+$, therefore we can apply the congruence rule for $+$ to get $\mathcal{AX}_{\mathcal{B}} \vdash t = u$.

Lemma 21. *$\forall t, u \in \mathcal{B}_{fin}, t, u$ in basic normal form: $t \sim_{\mathcal{B}} u \Rightarrow \mathcal{AX}_{\mathcal{B}} \vdash t = u$.*

Proof. With the proof for simple basic normal forms we can prove the completeness of $\mathcal{AX}_{\mathcal{B}}$ for *basic normal forms*. Let $t, u \in \mathcal{B}_{fin}$ in bnf with $t \sim_{\mathcal{B}} u$. We prove the theorem by induction over $d = depth(t) + depth(u)$.

- $d = 0$. We have to distinguish between the following possibilities:
  - $t = \sum\{\}$ and $u = \sum\{\}$. Therefore, the result follows immediately.
  - $t = spawn(t')$ and $u = spawn(u')$, t', u' in simple bnf. With $t \sim_{\mathcal{B}} u$ and $t'\checkmark, u'\checkmark$ we know that $t' \sim_{\mathcal{B}} u'$. By Lemma 20 we can deduce that $\mathcal{AX}_{\mathcal{B}} \vdash t' = u'$. $\sim_{\mathcal{B}}$ is a congruence for *spawn*, therefore we can apply the congruence rule for *spawn* and get $\mathcal{AX}_{\mathcal{B}} \vdash spawn(t') = spawn(u')$.
- $d > 0$. Then $t = \sum_{i\in\mathcal{I}} \alpha_i; t_i$ and $u = \sum_{k\in\mathcal{K}} \beta_k; u_k$. Analogous to the case in Lemma 20.

Proof of Theorem 8. With Theorem 7 we know that $\forall t \in \mathcal{B}_{fin} \exists u \in \mathcal{B}_{fin} : \mathcal{AX}_{\mathcal{B}} \vdash t = u$ and u is in basic normal form. Therefore, we can deduce in combination with Lemma 21 that Theorem 8 is valid for all terms $t \in \mathcal{B}_{fin}$.

□

B Proofs for the Full Calculus

Proposition 10: For every $t \in \mathcal{F}$, there is a unique $u \equiv t$ with u in *snf*.

We have to prove, that

- $\forall t \in \mathcal{F}_{\text{wf}} \exists u \in \mathcal{F}_{\text{wf}} : u \equiv t$ and u is in structural normal form (snf),
- if $u, v \in \mathcal{F}_{\text{wf}}$ in snf and $u \equiv t \wedge v \equiv t$, then $u = v$.

$\mathbf{1}; t \to t$	(41)	$t; (x : u) \to (x : t; u)$	(51)
$(t; u); v \to t; (u; v)$	(42)	$(x : t); u \to (x : t; u)$	(52)
$\sigma; (\mathbf{0}; v) \to (\mathbf{0}; \sigma); v$	(43)	$(\mathbf{x} : t; \sigma) \to t; (\sigma \setminus \{\mathbf{x}\})$ if $\{\mathbf{x}\} \subseteq \operatorname{dom}\sigma$ and $var(t) \cap \{\mathbf{x}\} = \emptyset$ and $\{\mathbf{x}\} \not\subseteq \operatorname{rng}\sigma$	(53)
$\sigma; (\mathbf{x}{\leftarrow}\mathbf{a}; v) \to (\sigma \cup \mathbf{x}{\leftarrow}\sigma(\mathbf{a})); v$	(44)	$\sigma; \mathbf{1} \to \sigma$	(54)
$\sigma; (x!y; v) \to (\sigma(x)!\sigma(y); \sigma); v$	(45)	$\sigma; \mathbf{0} \to \mathbf{0}; \sigma$	(55)
$\sigma; (x?y; v) \to (\sigma(x)?y; \sigma); v$	(46)	$\sigma; \mathbf{x}{\leftarrow}\mathbf{a} \to \sigma \cup \mathbf{x}{\leftarrow}\sigma(\mathbf{a})$	(56)
$\sigma; ([x{=}y]; v) \to ([\sigma(x){=}\sigma(y)]; \sigma); v$	(47)	$\sigma; x!y \to \sigma(x)!\sigma(y); \sigma$	(57)
$\sigma; (n(\mathbf{a}); v) \to (n(\sigma(\mathbf{a})); \sigma); v$	(48)	$\sigma; x?y \to \sigma(x)?y; \sigma$	(58)
$\sigma; (\mathit{spawn}(t); v) \to (\mathit{spawn}(\sigma; t); \sigma); v$	(49)	$\sigma; [x{=}y] \to [\sigma(x){=}\sigma(y)]; \sigma$	(59)
$\sigma; ((t + u); v) \to (\sigma; t + \sigma; u); v$	(50)	$\sigma; n(\mathbf{x}) \to n(\sigma(\mathbf{x})); \sigma$	(60)
		$\sigma; \mathit{spawn}(t) \to \mathit{spawn}(\sigma; t); \sigma$	(61)
		$\sigma; (t + u) \to \sigma; t + \sigma; u$	(62)

Fig. 7. Term rewriting system.

For the proof we use methods known from term rewriting [18]. We create a term rewriting system (TRS) for the equations of structural equivalence and show that this system is strongly normalizing and confluent and that the normal forms of the TRS are terms in snf. The TRS consists of the rewrite rules in Figure 7. We denote the applicability of rules in TRS to a given term t by $t \to_{TRS}$.

The TRS contains no rule for Equation (30), because we treat terms like $(x : (y : t))$ as $(\{x, y\} : t)$ (similarly t can be treated as $(\emptyset : t)$). The rules (43) to (50) have been added to the TRS in order to enable the reduction of terms of the form $\sigma; (t; u)$.

Lemma 22. *$\forall t \in \mathcal{F}_{\text{wf}} : t \not\to_{TRS}$ iff t is in snf or $t; \emptyset$ is in snf.*

Proof.
"⇒": We use a induction over the term structure.

- $t = \mathbf{0}$ or $t = \mathbf{1}$ or $t = a \dagger b$ or $t = [x{=}y]$ or $t = n(\mathbf{a})$. Then $t; \emptyset$ is in snf.
- $t = \mathit{spawn}(u)$. If u is not in snf, then $u \to_{TRS}$ by induction hypothesis, hence $t \to_{TRS}$. If u in snf, then $t; \emptyset$ is in snf.
- $t = (\mathbf{x} : u)$. Then we distingiush between the following cases:
  - u is not in snf. Then $u \to_{TRS}$ by induction hypothesis, hence $t \to_{TRS}$.
  - u is in snf, $u = (\mathbf{y} : u'; \sigma)$ and $\{\mathbf{x}\} \cap \operatorname{dom}\sigma \neq \emptyset$. Then we can apply rule (53).
  - u is in snf, $u = (\mathbf{y} : u'; \sigma)$ and $\{\mathbf{x}\} \cap \operatorname{dom}\sigma = \emptyset$. Then t is in snf.
- $t = u + v$. We distinguish between the following cases:
  - u is not in snf. Then $u \to_{TRS}$ by induction hypothesis.
  - v is not in snf. Then $v \to_{TRS}$ by induction hypothesis.
  - u, v are in snf. Then $t; \emptyset$ is in snf.
- $t = u; v$. We distinguish between the following cases:
  - u is not in snf. Then $u \to_{TRS}$ by induction hypothesis.
  - v is not in snf. Then $v \to_{TRS}$ by induction hypothesis.
  - u, v in snf and
    - $u = (x : u')$. Then we can apply rule (52).
    - $v = (x : v')$. Then we can apply rule (51).
    - $u = \sigma$ and $v \neq v'; v''$. Then we can apply the corresponding rule from rules (54) to (62).
    - $u = \sigma$ and $v = v'; v''$. Then we can apply the corresponding rule from rules (43) to (50).

$*$ $u = (u'; \sigma)$. Then we can apply rule (42).

"$\Leftarrow$": We show that none of the rules in TRS is applicable to a term t in snf.

- Rule (41) cannot be applied to t, because t must not contain $\mathbf{1}$.
- Rule (42) cannot be applied to t, because we assume sequential composition to be right associative. Therefore, sequential composition of subterms in t is right associative.
- Rules (43) to (50) and (54) to (62) cannot be applied to t, because in terms in snf only trailing substitutions must occur. All subterms in t must be in snf as well, therefore they contain only trailing substitutions, too.
- Rules (51) and (52) cannot be applied to t, because terms in snf must not contain subterms with surrounding restrictions in sequential compositions.
- Rule (53) cannot be applied to t, because with $t = (\mathbf{x} : u; \sigma)$ in snf we know that $\{\mathbf{x}\} \cap \operatorname{dom} \sigma = \emptyset$.

Lemma 23. *TRS is strongly normalizing, i.e.* $\forall t \in \mathcal{F}_{\text{wf}} \exists n \geq 0 \exists t_1, ..., t_n \in \mathcal{F}_{\text{wf}} : t \rightarrow_{TRS} t_1 \rightarrow_{TRS} ... \rightarrow_{TRS} t_n \wedge t_n \not\rightarrow_{TRS}$.

Proof. As a measure for the terms of $\mathcal{F}_{\text{wf}}$ we consider the number the of sequentially composed subterms that follow the left-most substitution in a given term. We show, that for every reduction rule $t \rightarrow u$ of Figure 7 this number for t is greater than or equal to the number for u. This means that substitutions are shifted from left to right through terms by the reduction rules. If the number on both sides is equal, we can use additional information.

Let $\mathit{seq\_ops} : \mathcal{F}_{\text{wf}} \rightarrow \mathbf{N}$ be a function which computes the number of sequentially composed subterms of a given term with right-associative sequential composition: $\mathit{seq\_ops}(t_1; (t_2; (...; t_n)...)) = n$. Furthermore, we define a function $\mathit{r\_assoc} : \mathcal{F}_{\text{wf}} \rightarrow \mathcal{F}_{\text{wf}}$, which transforms a given term t in a term t' in which sequential composition is right-associative.

$$\mathit{r\_assoc}(t) = \begin{cases} t & \text{if } \nexists t_1, t_2 : t = t_1; t_2 \\ \mathit{r\_assoc}(t_1; (t_2; t_3)) & \text{if } \exists t_1, t_2, t_3 : t = (t_1; t_2); t_3 \\ t_1; \mathit{r\_assoc}(t_2) & \text{if } \exists t_1; t_2 : t = t_1; t_2 \wedge \nexists u_1; u_2 : t_1 = u_2; u_3 \end{cases}$$

Next we define the partial function $\mathit{terms} : \mathcal{F}_{\text{wf}} \rightharpoonup \mathbf{N}$, which computes the the number of sequentially composed subterms in t up to and including the first substitution. The function is only defined for terms which contain substitutions.

$$\mathit{terms}(t) = \begin{cases} 1 & \text{if } t = \sigma \text{ or } \exists u : t = \sigma; u \\ 1 + \mathit{terms}(v) & \text{if } \exists u, v : t = u; v \wedge u \neq \sigma \end{cases}$$

As a fourth function we define a function $\mathit{restr} : \mathcal{F}_{\text{wf}} \rightarrow \mathbf{N}$, which computes the number of restricted variables in a given term: $\mathit{restr}((\{x_1, ..., x_n\} : t)) = n$.

With these functions we can define a function $\mathit{follow} : \mathcal{F}_{\text{wf}} \rightarrow \mathbf{N}$, which computes the number of subterms following the left-most substitution in a given term. If t does not contain substitutions, the function results 0.

$$\mathit{follow}(t) = \begin{cases} 0 & \text{if } t \text{ does not contain substitutions} \\ \mathit{seq\_ops}(t') - \mathit{terms}(t') & \text{if } t \text{ contains substitutions and } t' = \mathit{r\_assoc}(t) \end{cases}$$

The measure for the terms now can be defined as follows:

$$
\begin{aligned}
t < u \Leftrightarrow \quad & (\mathit{follow}(t) < \mathit{follow}(u)) \\
& \vee (\mathit{follow}(t) = \mathit{follow}(u) \wedge \mathit{seq\_ops}(t) < \mathit{seq\_ops}(u)) \\
& \vee (\mathit{restr}(t) < \mathit{restr}(u) \wedge \mathit{follow}(t) = \mathit{follow}(u) \wedge \mathit{seq\_ops}(t) = \mathit{seq\_ops}(u)) \\
& \vee (t = t_1; t_2 \wedge u = u_1; u_2 \wedge \mathit{seq\_ops}(t_1) < \mathit{seq\_ops}(u_1) \\
& \quad \wedge \mathit{seq\_ops}(t) = \mathit{seq\_ops}(u) \wedge \mathit{restr}(t) = \mathit{restr}(u) \wedge \mathit{follow}(t) = \mathit{follow}(u))
\end{aligned}
$$

We distinguish between the following cases (let $t \rightarrow u$ be the corresponding rule):

- For the rules (43) to (50) and (54) to (62) it is easy to verify, that $\mathit{follow}(t) > \mathit{follow}(u)$. Therefore, $t > u$.
- For the Equations (41), (42), (51), (52) and (53) $\mathit{follow}(t) = \mathit{follow}(u)$ holds.
  - For the Eqs. (41), (51) and (52) $\mathit{seq\_ops}(r\_\mathit{assoc}(t)) > \mathit{seq\_ops}(r\_\mathit{assoc}(u))$ holds.
  - In Equation (53) we have $\mathit{restr}(t) > \mathit{restr}(u)$.
  - $t = t_1; t_2, u = u_1; u_2$. Then we have in rule (42) that $\mathit{seq\_ops}(r\_\mathit{assoc}(t_1)) > \mathit{seq\_ops}(r\_\mathit{assoc}(u_1))$.

  Therefore, we can deduce that in all cases $t > u$ holds.

Next we have to show that the rewriting system TRS is confluent.

Lemma 24. *If $u, v \in \mathcal{F}_{\mathrm{wf}}$ in snf and $u \equiv t \wedge v \equiv t$, then $u = v$.*

Proof. To show the confluence property of the TRS, we can use the *critical pair lemma* from Knuth and Bendix (see e.g. Lemma 2.4.11 in [18]). In our TRS there exist only critical pairs which are convergent, i.e. if more than one rewrite rule is applicable to a given term, the corresponding reduction sequences result in the same normal form. Therefore, the TRS is weakly confluent. Additionally, by Lemma 23 we know that the TRS is strongly normalizing (every reduction sequence is terminating). From standard theory (see e.g. Lemma 2.4.14 in [18]) it is known that a strongly normalizing and weakly confluent TRS is confluent, therefore the TRS in Figure 7 is confluent.

Proof of Proposition 10. By the combination of Lemma 22, Lemma 23 and Lemma 24 we have proved Proposition 10.

□

Proposition 12. $\langle \mathcal{L}, \mathcal{F}_{\mathrm{wf}}, \rightarrow, \checkmark \rangle$ is an extended $\checkmark$-transition system.

Proof. We have to show that $\forall s, s' \in S$: if $s\checkmark_\sigma$ and $s \xrightarrow{a\dagger\xi} s'$ then $s'\checkmark_{\sigma\uparrow c_\xi}$. For this purpose, we have to consider the terminated terms in $\mathcal{F}_{\mathrm{wf}}$ which are able to perform actions. Let $t \in \mathcal{F}_{\mathrm{wf}}$ be a term with $t\checkmark_\sigma$ and $\exists t_{new} \in \mathcal{F}_{\mathrm{wf}} : t \xrightarrow{\alpha} t_{new}$. Therefore, we know that t must contain at least one *spawn*-operator. Let u be the structural normal form of t. With T_{10} we know that $u\checkmark_\sigma$. Furthermore, $u \xrightarrow{\alpha} u_{new}$ and $t_{new} \equiv u_{new}$, derived with R_{17}. We distinguish between the following cases:

1. $u = \mathit{spawn}(u')$ and $\sigma = \emptyset$. With R_6 we know that $\exists u'' : u_{new} = \mathit{spawn}(u'')$. With T_6 we know that $u\checkmark_\emptyset$ and $u_{new}\checkmark_\emptyset$. With $\emptyset = \emptyset \uparrow c_\xi$ the proposition holds.
2. $u = u_1; u_2; \sigma$. With u in snf we know that u_1, u_2 are in snf. Furthermore, with $u\checkmark_\sigma$ we know that u_1 and u_2 must equal $\mathbf{1}$ or $\mathit{spawn}(u')$ for some $u' \in \mathcal{F}_{\mathrm{wf}}$ (in the case $u_1 \in \mathbf{S}$ or $u_2 \in \mathbf{S}$ u would not be in snf). We distinguish between the following cases:

- Assume $u_1 = \mathit{spawn}(u_3)$ and $u_3 \xrightarrow{\alpha} u_3'$. Then $u_{new} = \mathit{spawn}(u_3'); u_2; \sigma$, derived with R_6 and R_7. If $r_\alpha \neq \emptyset$, then the effect of scope extrusion is restricted to u_3' and therefore does not influence the restriction behaviour of u_{new}. With $\forall u' \in \mathcal{F}_{\text{wf}} : \mathit{spawn}(u')\checkmark_\emptyset$ we can deduce that $u_{new}\checkmark_\sigma$. Therefore, the proposition holds.
- Assume $u_2 = \mathit{spawn}(u_4)$ and $u_4 \xrightarrow{\alpha} u_4'$. Then $u_{new} = u_1; \mathit{spawn}(u_4'); \sigma$, derived with R_6, R_7 and R_8. Analogous to the previous case.
- Assume $u_1 = \mathit{spawn}(u_3)$ and $u_2 = \mathit{spawn}(u_4)$ such that $u_3 \xrightarrow{\gamma} u_3'$, $u_4 \xrightarrow{\gamma'} u_4'$, $\{\gamma, \gamma'\} = \{a!\xi, a?\chi\}$ and $\alpha = \tau$; it follows (by R_{16}) that $u_{new} = (r_{\{\xi,\chi\}} : \{c_\chi \leftarrow c_\xi\}; \mathit{spawn}(u_3'); \mathit{spawn}(u_4'); \sigma)$, . By shifting $\{c_\chi \leftarrow c_\xi\}$ through the term we get $u'_{new} = (r_{\{\xi,\chi\}} : \mathit{spawn}(u_3''); \mathit{spawn}(u_4''); \{c_\chi \leftarrow c_\xi\} \circ \sigma)$. Due to the occurence of *spawn*-operations in u_1 and u_2 and the well-formedness of u'_{new} we know that $c_\chi \in r_{\{\xi,\chi\}}$, because c_χ must be restricted locally in the corresponding *spawn*-operation. To achieve the snf of u'_{new}, we can apply Eq. (33) to remove the restriction of r_χ and the substitution $\{c_\chi \leftarrow c_\xi\}$ from u'_{new} to get u''_{new}. If $c_\xi \in r_{\{\chi,\xi\}}$, then we know with the well-formedness of u''_{new} that $c_\xi \notin \sigma$. Therefore, the restriction of c_ξ does not influence the termination behaviour of u''_{new}. Then we know that u_{new} has snf $(r_\xi : \mathit{spawn}(u_3''); \mathit{spawn}(u_4''); \sigma)$ and therefore the proposition holds.

3. $u = (\mathbf{x} : u'; \sigma')$, $\sigma = \sigma' \downarrow \{\mathbf{x}\}$. With u in snf we know that $\{\mathbf{x}\} \cap \operatorname{dom}\sigma' = \emptyset$. We distinguish between the following cases:
   - Assume $u' \xrightarrow{\alpha} u_5$, $\alpha = a\dagger\xi$, $a \notin \{\mathbf{x}\}$ and $c_\xi \notin \{\mathbf{x}\}$. Then $u_{new} = (\mathbf{x} : u_5; \sigma')$, derived with R_7,R_{14}. If $r_\xi \neq \emptyset$, then we know with $c_\xi \notin \{\mathbf{x}\}$ that c_ξ was restricted locally in a *spawn*-operation. Therefore, we can deduce with the well-formedness of u_{new} that c_ξ does not occur in σ'. Then $\sigma' \uparrow c_\xi = \sigma'$ and $u''\checkmark_\sigma$. Therefore, the proposition holds.
   - Assume $u' \xrightarrow{\alpha} u_6$, $\alpha = a\dagger\xi$, $a \notin \{\mathbf{x}\}$ and $c_\xi \in \{\mathbf{x}\}$. Then $u_{new} = (\{\mathbf{x}\} \setminus c_\xi : u_6; \sigma')$, derived with R_7, R_{15}. Then $u_{new}\checkmark_{\sigma''}$ with $\sigma'' = \sigma' \downarrow (\{\mathbf{x}\} \setminus c_\xi)$. Therefore, $\sigma'' = \sigma \uparrow c_\xi$ and the proposition holds.

□