Flemming Nielson
Editor

# ML with Concurrency

## Design, Analysis, Implementation, and Application

With 18 Figures

Springer

# 7
# Communication Analysis for Concurrent ML

Hanne Riis Nielson and Flemming Nielson

ABSTRACT Concurrent ML (CML) is an extension of the functional language Standard ML (SML) with primitives for dynamic creation of processes and channels and for communication of values over channels. Because of the powerful abstraction mechanisms, the communication topology of a given program may be very complex and therefore an efficient implementation may be facilitated by knowledge of the topology.

This paper presents a framework for analyzing the communication topology of CML programs. We proceed by extending a polymorphic type system for SML to deduce not only the types of CML programs but also their communication *behaviors* expressed as terms in a process algebra. This involves a syntactic ordering for expressing when one behavior has more communication possibilities than another. We then provide an annotated version of the published operational semantics for CML and we provide an operational semantics for the behaviors similar to the operational semantics of other process algebras. Based on this we define a notion of simulation for behaviors and prove that the syntactic ordering is sound (but not complete). We then prove the *semantic correctness* of the *type and behavior inference system* by an extended subject reduction result: types of CML programs are preserved during evaluation whereas their behaviors may evolve as expressed by the semantics of behaviors. Finally we show that the simulation ordering is undecidable whereas the syntactic ordering (with the exclusion of the axiom for unfolding recursive behaviors) is decidable.

**Acknowledgment** This work is partially supported by ESPRIT BRA 8130 LOMAPS and by the DART-project funded by the Danish Science Research Council.

## 7.1 Introduction

In the previous chapters we have seen the design of a number of functional languages that incorporate constructs for achieving concurrency and distribution: CML [150, 151, 152] in Chapter 2, Poly/ML in Chapter 3, LCS [25] in Chapter 4, and FACILE [140] in Chapter 5. Despite the differences between the languages there are a number of similarities: they are all higher-order functional languages, they are all eager (as opposed to lazy)

languages, they allow the spawning of new processes and the creation of new channels, and they allow the sending and receiving of values over typed channels.

The design of these languages allows for a high degree of modularization and abstraction in programming but also has the undesirable consequence that the overall communication structure of a given program is not immediately clear from its structure. It is worth pointing out that similar phenomena arise when functional languages are extended with first-class continuations (or imperative languages are extended with goto's). This is somewhat unfortunate, despite the increased expressibility, because knowledge of the communication structure is essential for the implementation as well as for the programmer reasoning about the correctness of programs. To be more specific: as regards implementation, knowledge of the communication structure may facilitate processor allocation so as to match the network configuration; as regards the programmer, it may be important for him to establish that certain communication protocols are adhered to.

In Section 7.2 we therefore present an *analysis of the communication structure* of CML programs by extracting the communication behavior of the program. The development is inspired by the effect systems developed in [95, 160] for polymorphic type inference of functional languages with references. The approach of these papers has been modified to express the communications that take place during execution: in [167] an analysis is defined and in Chapter 6 the effects are used to define the denotational semantics of a concurrent functional language. However, these modifications retain the main characteristics of [95, 160] that effects are *sets* of individual actions. In contrast, the development to be performed here (building upon [122] and [125]) will retain the precise picture of the communication topology by including far more *causality* into the effects. This involves defining a syntactic ordering on behaviors for when one behavior has more communication possibilities than another; this then replaces the simple subset relation of [95, 160].

For the semantics in Section 7.3 we adapt the operational semantics of [151, 152] to include additional annotations that will be useful for our proofs; this technique was also used in [18] and is merely a convenient way of stating results without interfering with the dynamic properties of the semantics. Unlike [167] we then define an operational semantics of behaviors; this is in the spirit of operational semantics of process algebras. Based on this we define a notion of simulation on behaviors and we show that the syntactic ordering on behaviors is a sound (but not complete) axiomatization.

Unlike previous approaches to relating programming languages and process algebras, that tend to regard a programming language as a process algebra with value passing, we establish in Section 7.4 an extended *subject reduction result*; it says that types for CML programs are preserved during evaluation but that the behaviors may evolve as expressed by the operational semantics for behaviors.



We then show in Section 7.5 that the simulation ordering is undecidable by a reduction from language containment for simple grammars. We also show that the syntactic ordering is decidable (except for the inclusion of the axiom for unfolding recursive behaviors). Finally, we discuss the prospects and consequences of achieving decidability.

Our conclusions in Section 7.6 outline a few more specific analyses that can be built on top of the behaviors. Full details of the developments may be found in the appendices.

## 7.2 Extracting the Communication Topology

We shall follow [18, 152] and study a polymorphic subset of CML with *expressions* $e \in \mathbf{Exp}$ given by

$$\begin{array}{rcl} e & ::= & c \mid x \mid \texttt{fn}\ x \texttt{=>}\ e \mid e_1\ e_2 \\ & \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{rec}\ f\ x \texttt{=>}\ e \\ & \mid & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \end{array}$$

Here $x$ and $f$ are program identifiers. In addition to function abstraction and function application we have a polymorphic `let`-construct, recursion, and a conditional. The *constants* $c \in \mathbf{Const}$ are given by

$$\begin{array}{rcl} c & ::= & () \mid \texttt{true} \mid \texttt{false} \mid n \\ & \mid & \texttt{+} \mid \texttt{*} \mid \texttt{=} \mid \cdots \\ & \mid & \texttt{pair} \mid \texttt{fst} \mid \texttt{snd} \\ & \mid & \texttt{nil} \mid \texttt{cons} \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{isnil} \\ & \mid & \texttt{send} \mid \texttt{receive} \mid \texttt{choose} \\ & \mid & \texttt{wrap} \mid \texttt{sync} \mid \texttt{channel}_l \mid \texttt{fork}_\pi \end{array}$$

We have constants corresponding to the base types `unit`, `bool`, and `int` together with operations for constructing and destructing pairs and lists. We may send a value `v` over a channel `ch` by `sync(send(ch,v))`, receive a value over a channel `ch` by `sync(receive(ch))`, and choose between a list $[e_1, \cdots, e_n]$ of communications by `sync(choose(`$[e_1, \cdots, e_n]$`))`, where the case $n = 0$ is written `sync(noevent)` and acts as a blocking statement. Here the primitives `send`, `receive`, `choose`, and `noevent` do not actually perform the communications but produce *delayed communications* that are then *activated* by the `sync` operator. The operation `wrap`$(e_1, e_2)$ then modifies the delayed communication $e_1$ to another that applies $e_2$ to the resulting value; so `sync(wrap(`$e_1, e_2$`))` may be thought of as $e_2$`(sync(`$e_1$`))` provided that $e_2$ performs no communications. Finally, we may fork a process to the pool of processes and we may allocate a new free channel to be used for communication; as is clear from the syntax we shall assume that these primitives are annotated with labels $l, \pi$ from the set **Lab**.

**Example 7.2.1.** Consider the following CML program:

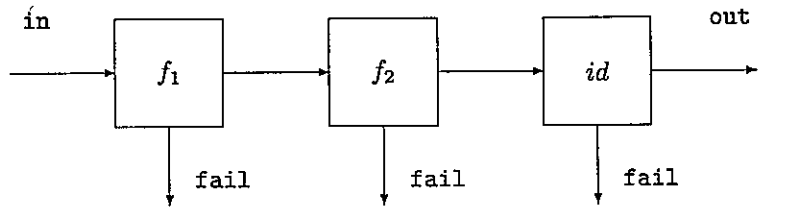


FIGURE 7.1. pipe $[f_1, f_2]$ in out

```
let node = fn f => fn in => fn out =>
    fork_π (rec loop d =>
          sync (choose [wrap (receive in,
                              fn x => sync (send (out, f x));
                                      loop d),
                        send(fail,())]))
in rec pipe fs => fn in => fn out =>
   if isnil fs
   then node (fn x => x) in out
   else let ch = channel_l ()
        in (node (hd fs) in ch; pipe (tl fs) ch out)
```

For the sake of readability we write $e_1;e_2$ for (fn dummy => $e_2$) $e_1$. Given a list of functions and two channels, the program will construct a pipeline of the functions using local channels for interconnecting the functions. Each of the functions may successfully be applied to its argument or cause a failure. This is illustrated in Figure 7.1 for a list with two elements. □

### 7.2.1 Types, Behaviors, and Regions

As usual we shall use *types* to classify the values that expressions can evaluate to. When executing a CML program, channels and processes may be created and values may be communicated, and we shall extend the type system with *behaviors* to record this. We do not know the identity of channels but can use the notion of *regions* to track the program points where a given channel could have been allocated.

For *types* $t \in$ **Typ** we take

$$\begin{array}{rcl} t & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \alpha \\ & \mid & t_1 \rightarrow^b t_2 \mid t_1 \times t_2 \mid t\ \texttt{list} \\ & \mid & t\ \texttt{chan}\ r \mid t\ \texttt{com}\ b \end{array}$$

Here $\alpha$ is a metavariable for *type variables*. The function type is written



$t_1 \rightarrow^b t_2$ indicating that the argument type is $t_1$, the result type is $t_2$, and the latent behavior is $b$; thus when a function is supplied with an argument the resulting behavior will be $b$. The type of a channel is $t$ chan $r$, indicating that the channel is allocated in region $r$ and that values of type $t$ can be communicated over it. Finally, $t$ com $b$ is the type of a *suspended* communication: when it is eventually enacted using sync, it will result in a value of type $t$ and the resulting behavior will be $b$.

Formally, *behaviors* $b \in$ **Beh** are given by

$$\begin{array}{rcl} b & ::= & \epsilon \mid r!t \mid r?t \mid t\ \textsc{chan}\ r \mid \beta \\ & \mid & \textsc{fork}_\pi\ b \mid b_1;b_2 \mid b_1 + b_2 \mid \textsc{rec}\ \beta.b \end{array}$$

Here $\epsilon$ stands for the nonobservable behavior; this will be the behavior associated with pure functional programs. We write $r!t$ for sending a value of type $t$ over a channel in region $r$ and similarly $r?t$ for receiving a value of type $t$ over a channel in region $r$. The allocation of a new channel in region $r$ is written $t$ CHAN $r$, where $t$ is the type of values to be communicated. The behavior $\textsc{fork}_\pi\ b$ expresses that a process with behavior $b$ and label $\pi$ is spawned. Behaviors may be combined using sequencing and choice and they may be recursive. We write $\beta$ for a metavariable for *behavior variables*. So for example REC $\beta$. ($t$ CHAN $r$+$\textsc{fork}_\pi(r?t;\ \beta)$) is the behavior of a program that either will create a channel, and then no more communications take place, or it will spawn a process that inputs on some channel and then the overall process repeats itself.

Finally, *regions* $r \in$ **Reg** are given by

$$r \quad ::= \quad l \mid r_1 + r_2 \mid \rho$$

Here $\rho$ denotes a metavariable for *region variables*. We shall think of + as *set union*, so a closed region will be a finite set of (channel) labels from **Lab**; formally, we define $r_1 \subseteq r_2$ to mean that all $l$'s and $\rho$'s appearing in $r_1$ also appear in $r_2$.

The *type schemes* are obtained from types by quantifying over type variables, behavior variables, and region variables: they have the form $\forall\vec{\alpha}\vec{\beta}\vec{\rho}.t$ where $\vec{\alpha}$, $\vec{\beta}$, and $\vec{\rho}$ are lists of variables.

As usual a type $t$ is a *generic instance* of a type scheme $ts = \forall\vec{\alpha}\vec{\beta}\vec{\rho}.t_0$, written $ts \succ t$, if there exists a substitution $\theta$ with $\mathrm{Dom}(\theta) = \{\vec{\alpha}\vec{\beta}\vec{\rho}\}$ such that $\theta\ t_0 = t$. Here a *substitution* $\theta$ is a finite mapping from type variables, behavior variables, and region variables to types, behaviors, and regions, respectively, and we write $\mathrm{Dom}(\theta)$ for its (finite) domain. Furthermore, a type scheme $ts'$ is an *instance* of $ts$, written $ts \succeq ts'$, if whenever $ts' \succ t$, also $ts \succ t$. See [50] for a different formulation that turns out to be equivalent [49].

We shall sometimes use $\gamma$ for any of $\alpha$, $\beta$, or $\rho$ and similarly use $\vec{\gamma}$ for $\vec{\alpha}\vec{\beta}\vec{\rho}$. Also we shall follow the conventions of the lambda-calculus [17] and freely perform alpha-renaming of bound behavior variables. As an example we have that $\texttt{int} \rightarrow^{\mathrm{REC}\beta_1.\beta_1} \texttt{int}$ equals $\texttt{int} \rightarrow^{\mathrm{REC}\beta_2.\beta_2} \texttt{int}$.

**Example 7.2.2.** The desired type of the function node of Example 7.2.1 is

$$\forall\alpha_1,\alpha_2,\beta,\rho_1,\rho_2.$$
$$(\alpha_1 \rightarrow^{\beta} \alpha_2) \rightarrow^{\epsilon} (\alpha_1 \text{ chan } \rho_1) \rightarrow^{\epsilon} (\alpha_2 \text{ chan } \rho_2) \rightarrow^{b} \texttt{unit}$$

where

$$b = \text{FORK}_{\pi}\ (\text{REC } \beta'.\ ((\rho_1?\alpha_1;\ \beta;\ \rho_2!\alpha_2;\ \beta') + \rho_0!\texttt{unit}))$$

Here we have assumed that `fail` has type `unit chan` $\rho_0$. Turning to the main program, the type is

$$\forall\alpha,\beta,\rho_1,\rho_2.$$
$$(\alpha \rightarrow^{\beta} \alpha)\ \texttt{list} \rightarrow^{\epsilon} (\alpha \text{ chan } (\rho_1 + l)) \rightarrow^{\epsilon} (\alpha \text{ chan } \rho_2) \rightarrow^{b'} \texttt{unit}$$

where

$$\begin{aligned} b' = \text{REC } \beta'.\ & (\text{FORK}_{\pi}\ (\text{REC } \beta''.\ (((\rho_1 + l)?\alpha;\ \rho_2!\alpha;\ \beta'') + \rho_0!\texttt{unit})) \\ + & (\alpha \text{ CHAN } (\rho_1 + l); \\ & \text{FORK}_{\pi}(\text{REC } \beta''.\ (((\rho_1 + l)?\alpha;\ \beta;\ \rho_2!\alpha;\ \beta'') + \rho_0!\texttt{unit})); \\ & \beta')) \end{aligned}$$

□

### *7.2.2 Ordering on Behaviors*

The behaviors record the communications taking place during evaluation. To obtain a flexible type system it is essential to be able to coerce a precise record of the behavior into a less precise record. This is illustrated in the following example:

**Example 7.2.3.** Consider the program

```
choose [send (ch, 7), wrap (receive ch', fn x => 1)]
```

where for the sake of readability we write `[`$e_1$`,` $e_2$`]` for `cons` $e_1$ `(cons` $e_2$ `nil)` and `(`$e_1$`,` $e_2$`)` for `pair` $e_1$ $e_2$. The first element of the list has type `int com r!int` (assuming `ch` has type `int chan r`) and the second element has type `int com r'?bool` (assuming `ch'` has type `bool chan r'`). We want the list to have type

```
(int com (r!int + r'?bool)) list
```

to record that either one of the branches may be chosen at run-time. So we need to coerce the types `int com r!int` and `int com r'?bool` into `int com (r!int + r'?bool)`. □

This leads to the introduction of a subsumption (or coercion) rule into the type and behavior inference system, and this presupposes an ordering on behaviors. We define the ordering $\sqsubseteq$ on behaviors by the axioms and rules of Figure 7.2 (to be explained below) and we write $\equiv$ for the associated equivalence defined by



- pre-order laws
  - **P1.** $b \sqsubseteq b$
  - **P2.** if $b_1 \sqsubseteq b_2$ and $b_2 \sqsubseteq b_3$ then $b_1 \sqsubseteq b_3$
- pre-congruence laws
  - **C1.** If $b_1 \sqsubseteq b_2$ and $b_3 \sqsubseteq b_4$ then $b_1; b_3 \sqsubseteq b_2; b_4$
  - **C2.** If $b_1 \sqsubseteq b_2$ and $b_3 \sqsubseteq b_4$ then $b_1 + b_3 \sqsubseteq b_2 + b_4$
  - **C3.** If $b_1 \sqsubseteq b_2$ then $\text{FORK}_{\pi}\ b_1 \sqsubseteq \text{FORK}_{\pi}\ b_2$
  - **C4.** If $b_1 \sqsubseteq b_2$ then $\text{REC } \beta.\ b_1 \sqsubseteq \text{REC } \beta.\ b_2$
- laws for sequencing
  - **S1.** $b_1; (b_2; b_3) \equiv (b_1; b_2); b_3$
  - **S2.** $(b_1 + b_2); b_3 \equiv (b_1; b_3) + (b_2; b_3)$
- laws for $\epsilon$
  - **E1.** $b \equiv \epsilon; b$
  - **E2.** $b; \epsilon \equiv b$
- laws for choice (or join)
  - **J1.** $b_1 \sqsubseteq b_1 + b_2$ and $b_2 \sqsubseteq b_1 + b_2$
  - **J2.** $b + b \equiv b$
- laws for recursion
  - **R1.** $\text{REC } \beta.\ b \equiv b[\beta \mapsto \text{REC } \beta.\ b]$
  - **R2.** $\text{REC } \beta.\ b \equiv \text{REC } \beta'.\ b[\beta \mapsto \beta']$ provided $\beta' \notin FV(b)$

FIGURE 7.2. Ordering on behaviors

$b_1 \equiv b_2$ if and only if $b_1 \sqsubseteq b_2$ and $b_2 \sqsubseteq b_1$.

We require $\sqsubseteq$ to be a pre-order and a pre-congruence. Furthermore, sequencing is an associative operation with $\epsilon$ as identity, and we have a left-distributive law with respect to choice. A consequence of the laws for choice is that choice is the least upper bound operator and hence associative and commutative. Finally, we have a couple of laws for recursion. It is always possible to fold and unfold a recursion (**R1**) and to rename the bound behavior variable (**R2**). Actually (**R2**) is implicitly true anyway given our syntactical conventions about alpha-renaming for bound behavior variables.

Unlike [122] we shall not extend the ordering $\sqsubseteq$ to types in order to avoid the complex interplay between polymorphism and subtyping.

**Remark.** The main difference between the typing system presented in this paper and those developed in [160, 167] is that in the present system the dependencies between the individual communications are recorded. If we were to extend Figure 7.2 with

$$b_1; b_2 \equiv b_1 + b_2$$

$$\text{REC } \beta.\ b \equiv b[\beta \mapsto \epsilon]$$

then our system would degrade to those of [29, 160, 167]. □

*Syntactic Properties of the Ordering*

For later reference we list a couple of properties holding for the ordering.

**Fact 7.2.4.** We have $b_1 \sqsubseteq b_2$ if and only if there exists $b$ such that $b_1 + b \equiv b_2$.

**Proof.** A simple consequence of the laws **C2**, **J1**, and **J2**. □

**Fact 7.2.5.** If $b_1 \sqsubseteq b_2$ then $FV(b_1) \subseteq FV(b_2)$.

**Proof.** A simple induction on the structure of the inference $b_1 \sqsubseteq b_2$. □

**Fact 7.2.6.** If $b_1 \sqsubseteq b_2$ and $\theta$ is a substitution then $\theta\ b_1 \sqsubseteq \theta\ b_2$.

**Proof.** A simple induction on the structure of the inference $b_1 \sqsubseteq b_2$. □

### 7.2.3 The Type and Behavior Inference System

We are now ready to develop the inference system for extracting types and behaviors. The *typing judgments* have the form

$$tenv \vdash e : t \ \&\ b$$

where *tenv* is a *type environment* mapping identifiers to types or type schemes, $t$ is the type of $e$, and $b$ is its behavior. Since CML has a call-by-value semantics, there is no observable behavior associated with accessing an identifier and therefore the type environment does not contain any behavior component (except embedded within the types or type schemes). The typing rules are given in Figure 7.3 and are fairly close to the standard ones except that we also collect behavior information; these rules will be explained in detail shortly.



$$tenv \vdash c : t \ \&\ b \qquad \text{if } \text{TypeOf}(c) \succ t \text{ and } \epsilon \sqsubseteq b$$

$$tenv \vdash x : t \ \&\ b \qquad \text{if } tenv(x) \succ t \text{ and } \epsilon \sqsubseteq b$$

$$\frac{tenv[x \mapsto t] \vdash e : t' \ \&\ b}{tenv \vdash \texttt{fn}\ x \texttt{=>}\ e : t \rightarrow^b t' \ \&\ b'} \qquad \text{if } \epsilon \sqsubseteq b'$$

$$\frac{tenv \vdash e_1 : t \rightarrow^b t' \ \&\ b_1 \qquad tenv \vdash e_2 : t \ \&\ b_2}{tenv \vdash e_1\ e_2 : t' \ \&\ b'} \qquad \text{if } b_1; b_2; b \sqsubseteq b'$$

$$\frac{tenv \vdash e_1 : t_1 \ \&\ b_1 \qquad tenv[x \mapsto ts] \vdash e_2 : t_2 \ \&\ b_2}{tenv \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2 \ \&\ b'}$$

$$\text{if } ts = \text{gen}(tenv, b_1)t_1 \text{ and } b_1; b_2 \sqsubseteq b'$$

$$\frac{tenv[f \mapsto t \rightarrow^b t'][x \mapsto t] \vdash e : t' \ \&\ b}{tenv \vdash \texttt{rec}\ f(x)\texttt{=>}e : t \rightarrow^b t' \ \&\ b'} \qquad \text{if } \epsilon \sqsubseteq b'$$

$$\frac{tenv \vdash e : \texttt{bool} \ \&\ b \qquad tenv \vdash e_1 : t \ \&\ b_1 \qquad tenv \vdash e_2 : t \ \&\ b_2}{tenv \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t \ \&\ b'}$$

$$\text{if } b; (b_1 + b_2) \sqsubseteq b'$$

FIGURE 7.3. Typing system

A main decision is how to incorporate the desired notion of subsumption that allows a precise record of the communication possibilities to be coerced into a less precise record. Basically, there are two approaches we may adopt:

- *Late subsumption*: coercions can happen at any time inside any type, as when the type system has a general subsumption rule on types. (In [162] this is called subtyping.)
- *Early subsumption*: generic instantiations produce the required specialized types. (In [162] this is called subeffecting.)

In [122] we used the first approach for a monotyped version of the language, thereby obtaining a type system with subtyping. Here we are in a polymorphic setting, and to avoid the complex interplay between polymorphism and subtyping we shall use the second approach (also taken in [125, 160]). This means that the latent behavior of functions and suspended communications always must be prepared to be larger than what seems to be needed.

The types of identifiers are obtained as generic instances of the appropriate type schemes. The actual behavior is $\epsilon$, but we may want to use a larger behavior, and to express this we exploit the ordering $\sqsubseteq$ on behaviors. This turns out to be a general pattern of the axioms and rules: it is always

possible to enlarge the actual behavior. In the rule for function abstraction we record the behavior of the body of the function as the latent behavior of the function type. The construction of a function does not in itself have an observable behavior and so is $\epsilon$. In the rule for function application we see that the actual behavior of the composite construct is that of the operator followed by that of the operand and then the behavior initiated by the function application itself; the latter is exactly the latent behavior of the function type. One may note that it is inherent in this rule that CML has a call-by-value semantics.

In the rule for local definitions we generalize over those type variables, behavior variables, and region variables that neither occur free in the type environment nor in the behavior; this is expressed by

$$\text{gen}(tenv, b)t = \text{let } \{\vec{\alpha}\vec{\beta}\vec{\rho}\} = FV(t) \setminus (FV(tenv) \cup FV(b)) \text{ in } \forall\vec{\alpha}\vec{\beta}\vec{\rho}.t,$$

where $FV(\cdots)$ denotes the set of free type variables, behavior variables, and region variables. The actual behavior of the `let`-construct simply expresses that the local value is computed before the body. In the rule for recursive functions we make sure that the actual behavior is equal to the latent behavior of the type of the recursive function. The rule for conditional should be straightforward; an alternative and equivalent definition would be to assume that the $b_1$ and $b_2$ occurring in the rule are equal (as may have been obtained by subsumption) and then use $b; b_1$ instead of $b; (b_1 + b_2)$.

This leaves us with the type schemes for constants. Each constant has associated a *constrained type scheme* as shown in Figure 7.4. A constrained type scheme $cts = \forall\vec{\alpha}\vec{\beta}\vec{\rho}.t_0[C]$ is a type scheme that additionally incorporates a constraint $C$; this is a finite set of inequalities of the form $\beta \geq b$ or $\rho \geq r$. A type $t$ is an instance of this constrained type scheme, written $cts \succ t$, if there exists a substitution $\theta$ with $\text{Dom}(\theta) = \{\vec{\alpha}\vec{\beta}\vec{\rho}\}$ such that $\theta\, t_0 = t$ and such that the constraints $C$ are solved by $\theta$, written $\theta \models C$. This latter condition amounts to $\theta\beta \sqsupseteq \theta b$ for each $\beta \geq b$ in $C$ and $\theta\rho \sqsupseteq \theta r$ for each $\rho \geq r$ in $C$. (Here $\geq$ is a formal symbol whereas $\sqsupseteq$ is the ordering defined in Figure 7.2.)

Note that for the primitives also to be found in SML the constraints only involve $\epsilon$, indicating that no communication need take place. Also, most of the primitives of CML are only constrained to have an $\epsilon$-annotation occur on the function arrows, although more interesting behaviors have to appear elsewhere in the type. The only three constants where function arrows are constrained to have non-$\epsilon$ behaviors are `sync`, which extracts the delayed communication of the argument and enacts it; `fork` which forks a new process; and `channel`, which allocates a new channel.

**Example 7.2.7.** Consider the program of Example 7.2.3, and let *tenv* be a type environment with *tenv* ch = `int chan r` and *tenv* ch′ = `bool chan r′`. By appropriate instantiations of the constrained type schemes of `send`



| $c$ | TypeOf($c$) |
|---|---|
| `+` | $\forall\beta_1,\beta_2.$ `int` $\rightarrow^{\beta_1}$ `int` $\rightarrow^{\beta_2}$ `int` $[\epsilon \leq \beta_1, \epsilon \leq \beta_2]$ |
| `pair` | $\forall\alpha_1,\alpha_2,\beta_1,\beta_2.\ \alpha_1 \rightarrow^{\beta_1} \alpha_2 \rightarrow^{\beta_2} \alpha_1 \times \alpha_2\ [\epsilon \leq \beta_1, \epsilon \leq \beta_2]$ |
| `fst` | $\forall\alpha_1,\alpha_2,\beta.\ \alpha_1 \times \alpha_2 \rightarrow^{\beta} \alpha_1\ [\epsilon \leq \beta]$ |
| `snd` | $\forall\alpha_1,\alpha_2,\beta.\ \alpha_1 \times \alpha_2 \rightarrow^{\beta} \alpha_2\ [\epsilon \leq \beta]$ |
| `send` | $\forall\alpha,\beta_1,\beta_2,\rho.\ (\alpha\,\texttt{chan}\,\rho) \times \alpha \rightarrow^{\beta_1} \alpha\,\texttt{com}\,\beta_2\ [\epsilon \leq \beta_1, \rho!\alpha \leq \beta_2]$ |
| `receive` | $\forall\alpha,\beta_1,\beta_2,\rho.\ (\alpha\ \texttt{chan}\ \rho) \rightarrow^{\beta_1} \alpha\ \texttt{com}\ \beta_2\ [\epsilon \leq \beta_1, \rho?\alpha \leq \beta_2]$ |
| `choose` | $\forall\alpha,\beta_1,\beta_2,\beta_3.\ (\alpha\,\texttt{com}\,\beta_1)\,\texttt{list} \rightarrow^{\beta_2} \alpha\,\texttt{com}\,\beta_3\ [\epsilon \leq \beta_2, \beta_1 \leq \beta_3]$ |
| `wrap` | $\forall\alpha_1,\alpha_2,\beta_1,\beta_2,\beta_3,\beta_4.$ $(\alpha_1\,\texttt{com}\,\beta_1) \times (\alpha_1 \rightarrow^{\beta_2} \alpha_2) \rightarrow^{\beta_3} \alpha_2\ \texttt{com}\ \beta_4$ $[\epsilon \leq \beta_3, \beta_1; \beta_2 \leq \beta_4]$ |
| `sync` | $\forall\alpha,\beta_1,\beta_2.\ (\alpha\ \texttt{com}\ \beta_1) \rightarrow^{\beta_2} \alpha\ [\beta_1 \leq \beta_2]$ |
| `channel`$_l$ | $\forall\alpha,\beta,\rho.$ `unit` $\rightarrow^{\beta} (\alpha\ \texttt{chan}\ \rho)\ [\alpha\ \text{CHAN}\ \rho \leq \beta, l \leq \rho]$ |
| `fork`$_\pi$ | $\forall\alpha,\beta_1,\beta_2.\ (\texttt{unit} \rightarrow^{\beta_1} \alpha) \rightarrow^{\beta_2} \texttt{unit}\ [\text{FORK}_\pi\ \beta_1 \leq \beta_2]$ |

FIGURE 7.4. Type schemes for selected constants

and `receive` we get

*tenv* ⊢ `send: int chan r × int` $\rightarrow^{\epsilon}$ `int com (r!int+r′?bool)` & $\epsilon$

*tenv* ⊢ `receive: bool chan r′` $\rightarrow^{\epsilon}$ `bool com (r!int+r′?bool)` & $\epsilon$

Using the typing rules of Figure 7.3 we then get

*tenv* ⊢ `send (ch, 7): int com (r!int + r′?bool)` & $\epsilon$

*tenv* ⊢ `wrap (receive ch′, fn x => 1)`

`: int com (r!int + r′?bool)` & $\epsilon$

so the two elements of the list get the same type, and therefore the overall typing of the program succeeds. □

*Syntactic Properties of the Typing System*

For later reference we list a couple of standard properties holding for the typing system; most are proved in Appendix 7.A.

**Lemma 7.2.8.** If $tenv[x \mapsto ts] \vdash e : t \,\&\, b$ and $ts' \succeq ts$ then $tenv[x \mapsto ts'] \vdash e : t \,\&\, b$.

**Fact 7.2.9.** If $tenv \vdash e : t \;\&\; b$ and $b \sqsubseteq b'$ then $tenv \vdash e : t \;\&\; b'$.

**Proof.** Inspection of the last step in the proof of $tenv \vdash e : t \;\&\; b$. □

**Lemma 7.2.10.** If $tenv \vdash e : t \;\&\; b$ and $\theta$ is a substitution, then $\theta\, tenv \vdash e : \theta\, t \;\&\; \theta\, b$.

We now define the set $IV(tenv \vdash e : t \& b)$ of independent variables occurring in the proof of $tenv \vdash e : t \& b$ to be all the variables in the proof tree except those of $FV(tenv)$.

**Fact 7.2.11.** Given a finite set $X$ of variables and an inference $tenv \vdash e : t \& b$ one may without loss of generality assume that $X \cap IV(tenv \vdash e : t \& b) = \emptyset$.

**Proof.** Let $\theta : IV(tenv \vdash e : t \& b) \to \overline{X}$ be a bijective renaming of all independent variables into the complement $\overline{X}$ of $X$; this is possible because $\overline{X}$ is infinite. Using Lemma 7.2.10 we get $(\theta\, tenv) \vdash e : (\theta\, t) \& (\theta\, b)$. By the definition of independent variables this amounts to $tenv \vdash e : (\theta\, t) \& (\theta\, b)$ and here we have no occurrences of variables from $X \setminus FV(tenv)$. □

**Lemma 7.2.12.** If $tenv \vdash e : t \& b$, $\mathrm{Dom}(tenv') \cap \mathrm{Dom}(tenv) = \emptyset$, and $FV(tenv') \cap IV(tenv \vdash e : t \& b) = \emptyset$ then $tenv\, tenv' \vdash e : t \& b$.

**Corollary 7.2.13.** If $tenv \vdash e : t \& b$ and $\mathrm{Dom}(tenv') \cap \mathrm{Dom}(tenv) = \emptyset$ then $tenv\, tenv' \vdash e : t \& b$.

**Lemma 7.2.14.** Assume $tenv[x \mapsto ts] \vdash e : t \;\&\; b$ and $tenv \vdash e_0 : t_0 \;\&\; \epsilon$. If $\mathrm{gen}(tenv, \epsilon)t_0 \succeq ts$ then $tenv \vdash e[x \mapsto e_0] : t \;\&\; b$.

### *7.2.4 Subject Expansion Properties*

To increase our faith in the typing rules for recursion and polymorphism it is instructive to compare the typing of the constructs with the typing of their unfolded versions.

The unfolded version of a recursively defined function can always be typed in the same way as the recursively defined function itself. To see this assume that

$$tenv \vdash \texttt{rec}\ f(x) \texttt{=>}\ e : t \to^b t' \;\&\; b',$$

because $\epsilon \sqsubseteq b'$ and $tenv[f \mapsto t \to^b t'][x \mapsto t] \vdash e : t' \;\&\; b$. But then

$$tenv[f \mapsto t \to^b t'] \vdash \texttt{fn}\ x \texttt{=>}\ e : t \to^b t' \;\&\; b'$$

as well as $tenv \vdash \texttt{rec}\ f(x) \texttt{=>}\ e : t \to^b t' \;\&\; \epsilon$. We now use Lemma 7.2.14 to obtain



$$tenv \vdash (\texttt{fn}\ x \texttt{=>}\ e)[f \mapsto \texttt{rec}\ f(x) \texttt{=>}\ e] : t \to^b t' \;\&\; b',$$

showing that the expansion $(\texttt{fn}\ x \texttt{=>}\ e)[f \mapsto \texttt{rec}\ f(x) \texttt{=>}\ e]$ has the same type and behavior as $\texttt{rec}\ f(x) \texttt{=>}\ e$.

For the `let`-construct things are more complicated, as shown by the following example:

**Example 7.2.15.** Consider the following expression $e$:

```
let x = sync (receive ch)
in sync (send (ch, 2)); x,
```

where we assume that `ch` has type `int chan r`. In the inference system we have

$$\cdots \vdash e : \texttt{int} \;\&\; \texttt{r?int;r!int}$$

The expansion of $e$ is

```
sync (send (ch, 2)); sync (receive ch)
```

and here the inference system gives

$$\cdots \vdash e : \texttt{int} \;\&\; \texttt{r!int;r?int}$$

The two behaviors `r?int;r!int` and `r!int;r?int` are incomparable, showing that the `let`-construct need not have the same behavior as its expansion. □

To obtain a positive result we consider the situation where the `let`-bound expression involves no communication. To be specific, assume that

$$tenv \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t \;\&\; b \quad \text{because} \quad tenv \vdash e_1 : t_1 \;\&\; \epsilon$$

and note that Lemma 7.2.14 then gives

$$tenv \vdash e_2[x \mapsto e_1] : t \;\&\; b,$$

so that in this case the expansion $e_2[x \mapsto e_1]$ has the same type and behavior as $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$.

## 7.3 Semantics

We shall now present a structural operational semantics for CML. The formulation is close in spirit to [152] and amounts to three inference systems: one for *sequential* evaluation, one for *concurrent* evaluation, and to handle synchronization we also need one for *matching* the communications against one another. This is mimicked in the specification of the semantics of behaviors where we have one inference system for *sequential* evolution and one for *concurrent* evolution. Matching is much simpler for behaviors than for programs, and therefore no matching relation is needed.

### 7.3.1 Semantics of CML

We begin with the *sequential evaluation* of expressions. This takes care of all primitives of CML except sync, $\texttt{channel}_l$, and $\texttt{fork}_\pi$, which are the constants of Figure 7.4 that have a nontrivial latent behavior. The transition relation for sequential evaluation has the form

$$e \rightarrow e',$$

where $e$ and $e'$ are *closed* expressions, that is, they do not contain free program identifiers. To enforce a left-to-right evaluation we introduce the concept of an *evaluation context* $E$ [55, 182], which specifies where the next step of the computation may take place:

$$E ::= [\,] \mid E\,e \mid w\,E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2.$$

Here $w$ denotes a weakly evaluated expression [142] (see below), that is, an expression that cannot be further evaluated. The idea is that $[\,]$ is an empty context (called a hole), and in general $E$ specifies a context with exactly one hole in it. We shall then write $E[e]$ for the expression $E$ with the hole replaced by $e$. The next step of the computation will take place at the point indicated by the hole. As an example consider function application. The presence of $E\,e$ means that computations in the operator position are possible whereas the presence of $w\,E$ means that computations in the operand position are possible only when the operator is weakly evaluated (for example to a function abstraction). In this way it is ensured that the operator as well as the operand are evaluated before the function application itself takes place (for example by $\beta$-reduction).

The *weakly evaluated expressions* $w \in \mathbf{WExp}$ are given by

$$w ::= c' \mid x \mid \texttt{fn}\ x \Rightarrow e \mid \langle c'\,w_1\rangle \mid \ldots \mid \langle c'\,w_1 \ldots w_n\rangle,$$

where $n \geq 1$ and $c'$ ranges over all constants except sync, $\texttt{channel}_l$, and $\texttt{fork}_\pi$. Weakly evaluated expressions of the form $\langle c'\,w_1 \ldots w_i\rangle$ are used to record the evaluation of constants as indicated in Figure 7.5, where we define the relation $\delta \subseteq \mathbf{WExp} \times \mathbf{WExp} \times \mathbf{WExp}$. Note that $\delta$ is partial, so for example hd nil is undefined.

The transition relation is specified in Figure 7.6. The clauses should be fairly straightforward. The first rule expresses the one-level unfolding of a recursive definition. Then we have axioms for $\beta$-reduction and for let-reduction. The fourth axiom is an abbreviation for two axioms expressing the evaluation of a conditional depending on the outcome of the test. Finally, there is an axiom for $\delta$-reduction which inspects Figure 7.5 to determine the result.

We shall now introduce the transition relation for *concurrent evaluation*. Channels will be associated with *channel identifiers*, $ci \in \mathbf{CIdent}$, and processes with *process identifiers*, $pi \in \mathbf{PIdent}$. We shall assume that the sets **CIdent**, **PIdent**, and **Ident** (of program identifiers) are mutually



| Operator | Operand | Result |
|---|---|---|
| pair | $w_1$ | $\langle \texttt{pair}\ w_1\rangle$ |
| $\langle \texttt{pair}\ w_1\rangle$ | $w_2$ | $\langle \texttt{pair}\ w_1\ w_2\rangle$ |
| fst | $\langle \texttt{pair}\ w_1\ w_2\rangle$ | $w_1$ |
| snd | $\langle \texttt{pair}\ w_1\ w_2\rangle$ | $w_2$ |
| cons | $w_1$ | $\langle \texttt{cons}\ w_1\rangle$ |
| $\langle \texttt{cons}\ w_1\rangle$ | $w_2$ | $\langle \texttt{cons}\ w_1\ w_2\rangle$ |
| hd | $\langle \texttt{cons}\ w_1\ w_2\rangle$ | $w_1$ |
| tl | $\langle \texttt{cons}\ w_1\ w_2\rangle$ | $w_2$ |
| isnil | nil | true |
| isnil | $\langle \texttt{cons}\ w_1\ w_2\rangle$ | false |
| send | $w$ | $\langle \texttt{send}\ w\rangle$ |
| receive | $w$ | $\langle \texttt{receive}\ w\rangle$ |
| choose | $w$ | $\langle \texttt{choose}\ w\rangle$ |
| wrap | $w$ | $\langle \texttt{wrap}\ w\rangle$ |
| + | $n_1$ | $\langle +\ n_1\rangle$ |
| $\langle +\ n_1\rangle$ | $n_2$ | $n$ where $n = n_1 + n_2$ |
| $\vdots$ | $\vdots$ | $\vdots$ |

FIGURE 7.5. Tabulation of $\delta$

$$
\begin{aligned}
E[\,\texttt{rec}\ f(x) \Rightarrow e\,] &\rightarrow E[\,(\texttt{fn}\ x \Rightarrow e)\,[f \mapsto (\texttt{rec}\ f(x) \Rightarrow e)]] \\
E[\,(\texttt{fn}\ x \Rightarrow e)\ w\,] &\rightarrow E[\,e\,[x \mapsto w]\,] \\
E[\,\texttt{let}\ x = w\ \texttt{in}\ e\,] &\rightarrow E[\,e\,[x \mapsto w]\,] \\
E[\,\texttt{if}\ w\ \texttt{then}\ e_1\ \texttt{else}\ e_2\,] &\rightarrow \begin{cases} E[e_1] & \text{if } w = \texttt{true} \\ E[e_2] & \text{if } w = \texttt{false} \end{cases} \\
E[\,w_1\,w_2\,] &\rightarrow E[w_3] \quad \text{if } (w_1, w_2, w_3) \in \delta
\end{aligned}
$$

FIGURE 7.6. Sequential evaluation

$$\frac{E[e] \to E[e']}{CI \,\&\, PP[pi \mapsto E[e]] \longrightarrow^{\epsilon}_{pi} CI \,\&\, PP[pi \mapsto E[e']]}$$

$$CI \,\&\, PP[pi \mapsto E[\texttt{channel}_l\ ()]] \longrightarrow^{\text{CHAN}_l\ ci}_{pi} CI \cup \{ci\} \,\&\, PP[pi \mapsto E[ci]]$$
$$\text{if } ci \notin CI$$

$$CI \,\&\, PP[pi_1 \mapsto E[\texttt{fork}_\pi\, w]] \longrightarrow^{\text{FORK}_\pi\ pi_2}_{pi_1,pi_2} CI \,\&\, PP[pi_1 \mapsto E[()]][pi_2 \mapsto w()]$$
$$\text{if } pi_2 \notin \text{Dom}(PP) \cup \{pi_1\}$$

$$\frac{(w_1, w_2) \overset{(ci!,ci?)}{\rightsquigarrow} (e_1, e_2)}{CI \,\&\, PP[pi_1 \mapsto E_1[\texttt{sync}\, w_1]][pi_2 \mapsto E_2[\texttt{sync}\, w_2]] \longrightarrow^{(ci!,ci?)}_{pi_1,pi_2} CI \,\&\, PP[pi_1 \mapsto E_1[e_1]][pi_2 \mapsto E_2[e_2]]}$$
$$\text{if } pi_1 \neq pi_2$$

FIGURE 7.7. Concurrent evaluation

disjoint. The configurations have the form $CI \,\&\, PP$, where $CI$ is the set of channel identifiers that are in use and $PP$ is a (finite) mapping of process identifiers to expressions. The transition relation is written

$$CI \,\&\, PP \longrightarrow^{ev}_{ps} CI' \,\&\, PP',$$

where $ev$ is the event that takes place and $ps$ is a list of the processes that take part in the event—depending on the event there will be either one or two processes involved. An *event* $ev \in \mathbf{Ev}$ has one of the forms

$$ev ::= \epsilon \mid \text{CHAN}_l\ ci \mid \text{FORK}_\pi\ pi \mid (ci!, ci?)$$

and may record the empty event, the creation of a channel with a given channel identifier, the creation of a process with a given process identifier, and the communication over a channel.

The transition relation is specified in Figure 7.7. The first rule embeds sequential evaluation into concurrent evaluation and the name of the process performing the event is recorded. The second rule captures the creation of a new channel. The channel is associated with a new channel identifier and the transition records the name of the process performing the event together with the event itself. The third rule takes care of process creation, and here we record the process performing the event as well as the one being created by the event. Finally, we have a rule expressing the synchronization of communications, and here we use the matching relation (explained below). The transition records the two processes involved in the communication as well as the channel used for it.



$$(\langle \texttt{send}\langle \texttt{pair}\ ci\ w\rangle\rangle, \langle \texttt{receive}\ ci\rangle) \overset{(ci!,ci?)}{\rightsquigarrow} (w, w)$$

$$\frac{(w_1, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_1, e_3)}{(\langle \texttt{choose}\langle \texttt{cons}\, w_1\, w_2\rangle\rangle, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_1, e_3)}$$

$$\frac{(\langle \texttt{choose}\, w_2\rangle, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_2, e_3)}{(\langle \texttt{choose}\langle \texttt{cons}\, w_1\, w_2\rangle\rangle, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_2, e_3)}$$

$$\frac{(w_1, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_1, e_3)}{(\langle \texttt{wrap}\langle \texttt{pair}\, w_1\, w_2\rangle\rangle, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (w_2\, e_1, e_3)}$$

$$\frac{(w_1, w_2) \overset{(d_1,d_2)}{\rightsquigarrow} (e_1, e_2)}{(w_2, w_1) \overset{(d_2,d_1)}{\rightsquigarrow} (e_2, e_1)}$$

FIGURE 7.8. Matching communications

Finally, the *matching relation* is given two weakly evaluated expressions that are ready to synchronize, and it specifies the outcome of the communication and records the event that takes place. This is expressed by a relation of the form

$$(w_1, w_2) \overset{(ci!,ci?)}{\rightsquigarrow} (e_1, e_2) \text{ and } (w_1, w_2) \overset{(ci?,ci!)}{\rightsquigarrow} (e_1, e_2)\ .$$

The relation is specified in Figure 7.8. The first axiom captures the communication between a `send` and a `receive` construct. The second and third axioms take care of the situation where there are several possible communications available in the first component. The fourth axiom shows how the value communicated may be modified using the `wrap` construct (as was explained already in Section 7.2). Finally, we have a restructuring rule.

### 7.3.2 *Semantics of Behaviors*

We begin with the *sequential evolution* of behaviors. Here the configurations of the transition system are either *closed* behaviors, that is, behaviors without free behavior variables, or the special *terminating configuration* $\surd$. The transition relation takes the form

$$b \Rightarrow^{p} \flat,$$

where $\flat$ is either a closed behavior or $\surd$, and where $p \in \mathbf{ABeh}$ is an *atomic behavior* as given by

$$p \quad ::= \quad \epsilon \mid r!t \mid r?t \mid t\ \text{CHAN}\ r \mid \text{FORK}_\pi\ b.$$

$$p \Rightarrow^{p} \epsilon \qquad \epsilon \Rightarrow^{\epsilon} \surd$$

$$b \Rightarrow^{\epsilon} b \qquad \text{REC } \beta.\ b \Rightarrow^{\epsilon} b[\beta \mapsto \text{REC } \beta.\ b]$$

$$\frac{b_1 \Rightarrow^{p} b_1'}{b_1; b_2 \Rightarrow^{p} b_1'; b_2} \qquad \frac{b_1 \Rightarrow^{p} \surd}{b_1; b_2 \Rightarrow^{p} b_2}$$

$$\frac{b_1 \Rightarrow^{p} b_1'}{b_1 + b_2 \Rightarrow^{p} b_1'} \qquad \frac{b_2 \Rightarrow^{p} b_2'}{b_1 + b_2 \Rightarrow^{p} b_2'}$$

FIGURE 7.9. Sequential evolution

Here $\epsilon$ is supposed to capture the sequential evaluation steps of CML expressions, whereas the remaining atomic behaviors capture the concurrent steps.

The relation is specified in Figure 7.9. The first axiom expresses that any atomic behavior can be performed and in doing so becomes $\epsilon$. The second axiom expresses that $\epsilon$ can terminate. The third axiom means that at any time any number of $\epsilon$ actions can be performed by any behavior (observe that the terminal configuration is excluded here so that $\surd$ is a stuck configuration). It corresponds to the fact that in the semantics of CML any number of evaluation steps can be performed in the functional part of CML between those involving the concurrency primitives. The fourth axiom expresses the unfolding of a recursive behavior. Then we have two rules for the evolution of sequential behaviors: only when the evolution of the first component has reached a terminal configuration is it possible to start evolution of the second component. The last two rules express the evolution of a choice between two behaviors; due to the axiom $b \Rightarrow^{\epsilon} b$ these rules allow *internal choice* between possibilities.

To express the *concurrent evolution* of behaviors we introduce *process identifiers* as in the semantics of CML. The transitions have the form

$$PB \Longrightarrow^{a}_{ps} PB',$$

where $PB$ and $PB'$ are mappings from process identifiers to closed behaviors and the special symbol $\surd$. Furthermore, $a$ is the action that takes place and $ps$ is a list of the processes that take part in the action. As in the semantics of CML, $ps$ has one or two elements depending on the action. The *actions* $a \in \mathbf{Act}$ are given by

$$a ::= \epsilon \mid t \text{ CHAN } r \mid \text{FORK}_{\pi}\ b \mid (r!t, r?t).$$

The transition relation is specified in Figure 7.10. The first four rules embed sequential evolution into the concurrent evolution: the first rule captures the termination of a behavior, the second rule captures a silent action, the third rule captures channel creation, and the fourth rule captures process creation. In all cases the action as well as the processes involved are



$$\frac{b \Rightarrow^{\epsilon} \surd}{PB[pi \mapsto b] \Longrightarrow^{\epsilon}_{pi} PB[pi \mapsto \surd]}$$

$$\frac{b \Rightarrow^{\epsilon} b'}{PB[pi \mapsto b] \Longrightarrow^{\epsilon}_{pi} PB[pi \mapsto b']}$$

$$\frac{b \Rightarrow^{t \text{ CHAN } r} b'}{PB[pi \mapsto b] \Longrightarrow^{t \text{ CHAN } r}_{pi} PB[pi \mapsto b']}$$

$$\frac{b \Rightarrow^{\text{FORK}_{\pi}\ b_0} b'}{PB[pi_1 \mapsto b] \Longrightarrow^{\text{FORK}_{\pi}\ b_0}_{pi_1, pi_2} PB[pi_1 \mapsto b'][pi_2 \mapsto b_0]} \quad \text{if } pi_2 \notin \text{Dom}(PB) \cup \{pi_1\}$$

$$\frac{b_1 \Rightarrow^{r!t} b_1' \qquad b_2 \Rightarrow^{r?t} b_2'}{PB[pi_1 \mapsto b_1][pi_2 \mapsto b_2] \Longrightarrow^{(r!t, r?t)}_{pi_1, pi_2} PB[pi_1 \mapsto b_1'][pi_2 \mapsto b_2']} \quad \text{if } pi_1 \neq pi_2$$

FIGURE 7.10. Concurrent evolution

recorded. The final rule captures the communication between processes. Here the matching simply amounts to ensuring that the channels of the two processes are in *equal* regions and that they specify *equal* types of the values communicated.

### 7.3.3 *Simulation Ordering on Behaviors*

In order to formulate and prove the subject reduction result in the next section we need to relate the ordering $\sqsubseteq$ on behaviors to the semantics of behaviors. Basically this amounts to the definition of a simulation ordering on behaviors and a proof showing that $\sqsubseteq$ is a simulation ordering. First define

$$b \Rightarrow\!\!\!\!\!\!\widehat{\ }^{p}\ \flat$$

to mean that there exists $n \geq 0$ and behaviors $b_1, \cdots, b_n$ such that

$$b \Rightarrow^{\epsilon} b_1 \Rightarrow^{\epsilon} \cdots \Rightarrow^{\epsilon} b_n \Rightarrow^{p} \flat.$$

Recall that $\flat$ ranges over closed behaviors as well as $\surd$. Thus the atomic behavior $p$ may be prefixed by any number of trivial atomic behaviors.

We shall say that $\mathcal{S}$ is a *ground simulation* on pairs of closed behaviors if

- $\surd\ \mathcal{S}\ \flat$ if and only if $\flat = \surd$,
- if $b_1 \Rightarrow^{p_1} \flat_1$ and $b_1\ \mathcal{S}\ b_2$ then there exists $\flat_2$ and $p_2$ such that $b_2 \Rightarrow\!\!\!\!\!\!\widehat{\ }^{p_2}\ \flat_2$, $p_1\ \mathcal{S}^{\theta}\ p_2$ and $\flat_1\ \mathcal{S}\ \flat_2$,

where $\mathcal{S}^\partial$ is defined by

- $p\ \mathcal{S}^\partial\ p$ whenever $p$ is a primitive action of the form $\epsilon, r!t, r?t$ or $t$ CHAN $r$, and
- $(\text{FORK}_\pi\ b_1)\ \mathcal{S}^\partial\ (\text{FORK}_\pi\ b_2)$ if $b_1\ \mathcal{S}\ b_2$.

We shall say that $\mathcal{S}$ is a *simulation* on pairs of behaviors $(b_1, b_2)$ satisfying $FV(b_1) \subseteq FV(b_2)$ if

- $b_1\ \mathcal{S}\ b_2$ implies $(\theta\ b_1)\ \mathcal{S}\ (\theta\ b_2)$

  for all ground substitutions $\theta$ defined on $FV(b_1) \cup FV(b_2)$.

We define $\precsim$ to be the *largest* simulation.

**Fact 7.3.1.** If $FV(b_1) \subseteq FV(b_2)$ then $b_1 \precsim b_2$ is equivalent to $(\theta\ b_1) \precsim (\theta\ b_2)$ for all ground substitutions $\theta$ defined on $FV(b_1) \cup FV(b_2)$.

**Proof.** Define $\precsim'$ by the property claimed for $\precsim$: $b_1 \precsim' b_2$ if and only if $(\theta\ b_1) \precsim (\theta\ b_2)$ for all ground substitutions $\theta$ defined on $FV(b_1) \cup FV(b_2)$. Clearly $\precsim'$ is a simulation so $b_1 \precsim' b_2$ implies $b_1 \precsim b_2$ by the choice of $\precsim$. To see that $b_1 \precsim b_2$ implies $b_1 \precsim' b_2$ simply use that $\precsim$ and $\precsim'$ agree on closed behaviors, that $\precsim$ is a simulation, and the definition of $\precsim'$. □

This definition of simulation is inspired by the notions of bisimilarity as developed for the process algebras CCS [108] and CHOCS [164].

In order to relate $\precsim$ and $\sqsubseteq$ we extend the ordering $\sqsubseteq$ to configurations by taking $\surd \sqsubseteq \surd$. Then we have the following results:

**Proposition 7.3.2.** $\sqsubseteq$ is a simulation.

**Proof.** See Appendix 7.B □

**Corollary 7.3.3.** If $b_1 \sqsubseteq b_2$ then $b_1 \precsim b_2$.

Corollary 7.3.3 shows that $\sqsubseteq$ as defined in Figure 7.2 is a *sound axiomatization* of $\precsim$. It is *not* complete because for $FV(b_1) \subseteq FV(b_2)$ we always have $(\text{REC}\beta.\beta); b_1 \precsim b_2$ whereas $(\text{REC}\beta.\beta); b_1 \sqsubseteq b_2$ does not hold in general given the axioms and rules of Figure 7.2. Similarly we have $\text{REC}\beta.(\beta + b) \precsim \text{REC}\beta.b$ whereas we do not have $\text{REC}\beta.(\beta + b) \sqsubseteq \text{REC}\beta.b$.

**Remark.** For certain applications it might be worthwhile to introduce laws like

$$t_1 \text{ CHAN } r_1;\ t_2 \text{ CHAN } r_2 \sqsubseteq t_2 \text{ CHAN } r_2;\ t_1 \text{ CHAN } r_1,$$

$$\text{FORK}_{\pi_1}\ b_1;\ \text{FORK}_{\pi_2}\ b_2 \sqsubseteq \text{FORK}_{\pi_2}\ b_2;\ \text{FORK}_{\pi_1}\ b_1,$$

reflecting that the order in which channel identifiers and process identifiers are allocated does not matter. However, these axioms are *not* sound with respect to our definition of simulation. Overcoming this is no easy task; the problems are analogous to those arising for imperative languages in trying to ensure that the order of declarations of identifiers is immaterial. □



## 7.4 Subject Reduction Property

We shall prove that the typing system of Section 7.2 has the following subject reduction properties:

- Types are preserved during computation.
- Behaviors evolve during computation.

The formalization and proof of this result is in three main stages: *(i)* We prove a subject reduction property for the sequential evaluation of expressions; *(ii)* then we prove a correctness property for matching; *(iii)* and finally we prove the subject reduction property for concurrent evaluation.

*Further Syntactic Properties of the Typing System*

But first it is convenient to establish a few additional syntactic properties of the inference system that concern the properties of evaluation contexts. Most of these results are proved in Appendix 7.C.

**Fact 7.4.1.** If $tenv \vdash E[e_0] : t\,\&\,b$ because $tenv_0 \vdash e_0 : t_0\,\&\,b_0$ then $tenv = tenv_0$ and $FV(b) \supseteq FV(b_0)$; hence $FV(tenv) \cup FV(b) \supseteq FV(tenv_0) \cup FV(b_0)$.

**Proof.** This is a straightforward induction on $E$. □

**Lemma 7.4.2.** If $tenv \vdash E[e_0] : t\,\&\,b$ because $tenv \vdash e_0 : t_0\,\&\,b_0$, and if $FV(tenv') \subseteq FV(b_0)$ and $\text{Dom}(tenv')$ is disjoint with all identifiers occurring in $tenv \vdash E[e_0] : t\,\&\,b$, then $tenv\,tenv' \vdash E[e_0] : t\,\&\,b$ because $tenv\,tenv' \vdash e_0 : t_0\,\&\,b_0$.

**Proof.** This is proved by induction on $E$ in Appendix 7.C. □

**Fact 7.4.3.** Assume that $tenv \vdash E[e_0] : t\ \&\ b$ because $tenv \vdash e_0 : t_0\ \&\ b_0$; if $tenv \vdash e_0' : t_0\ \&\ b_0$ then $tenv \vdash E[e_0'] : t\ \&\ b$ because $tenv \vdash e_0' : t_0\ \&\ b_0$.

**Lemma 7.4.4.** Assume that

$$tenv \vdash E[e_0] : t\ \&\ b \qquad \text{because} \qquad tenv \vdash e_0 : t_0\ \&\ b_0$$

If furthermore $tenv \vdash e_0' : t_0 \;\&\; b_0'$ where $p; b_0' \sqsubseteq b_0$ for some atomic behavior $p$ then there exists $b'$ such that

$$tenv \vdash E[e_0'] : t \;\&\; b'$$

and $p; b' \sqsubseteq b$.

### 7.4.1 Sequential Correctness

It is natural to restrict attention to closed expressions because the definition of an evaluation context is such that we never pass inside the scope of any defining occurrence of a program identifier. However, we have to allow expressions to include channel identifiers that have been allocated in previous computation steps. To formalize this we shall write *cenv* for a mapping from channel identifiers to types (so *cenv ci* will always have the form $t$ chan $r$). We shall say that $e$ is *closed* if $cenv \vdash e : t \;\&\; b$ for some *cenv*, $t$ and $b$; this requires the type environments of Figure 7.3 to range over channel identifiers as well as program identifiers. To express the correctness result we shall also need typing rules for weakly evaluated expressions:

$$\frac{tenv \vdash c' : t \to^{\epsilon} t' \;\&\; \epsilon \qquad tenv \vdash w_1 : t \;\&\; \epsilon}{tenv \vdash \langle c'\ w_1 \rangle : t' \;\&\; b} \quad \text{if } \epsilon \sqsubseteq b$$

$$\frac{tenv \vdash \langle c'\ w_1 \cdots w_{n-1} \rangle : t \to^{\epsilon} t' \;\&\; \epsilon \qquad tenv \vdash w_n : t \;\&\; \epsilon}{tenv \vdash \langle c'\ w_1 \cdots w_n \rangle : t' \;\&\; b} \quad \text{if } \epsilon \sqsubseteq b$$

It is immediate from these rules that we have

**Fact 7.4.5.** If $tenv \vdash w : t \;\&\; b$ then $\epsilon \sqsubseteq b$ and $tenv \vdash w : t \;\&\; \epsilon$.

We now have the following result showing that sequential evaluation preserves the type and behavior:

**Proposition 7.4.6.** Assume $e \to e'$ and $cenv \vdash e : t \;\&\; b$. Then $cenv \vdash e' : t \;\&\; b$.

**Proof.** The proof is by induction on the inference $e \to e'$ and is given in Appendix 7.C. □

### 7.4.2 Correctness of Matching

The matching of two weakly evaluated expressions gives rise to a new pair of expressions. To formalize this we shall define $\mathcal{B}\ cenv\ ci! = r!t$ and $\mathcal{B}\ cenv\ ci? = r?t$ whenever $cenv\ ci = t$ chan $r$. Then we have the following result showing how behaviors evolve into an atomic behavior and the remaining part of the behavior:



**Proposition 7.4.7.** Assume $(w_1, w_2) \overset{(d_1,d_2)}{\rightsquigarrow} (e_1, e_2)$ and

$$cenv \vdash w_1 : t_1 \text{ com } b_1 \;\&\; \epsilon \text{ and } cenv \vdash w_2 : t_2 \text{ com } b_2 \;\&\; \epsilon.$$

Then there exists $b_1'$ and $b_2'$ such that

$$cenv \vdash e_1 : t_1 \;\&\; b_1' \text{ and } cenv \vdash e_2 : t_2 \;\&\; b_2'$$

and $(\mathcal{B}\ cenv\ d_1); b_1' \sqsubseteq b_1$ and $(\mathcal{B}\ cenv\ d_2); b_2' \sqsubseteq b_2$.

**Proof.** The proof is by induction on the inference for matching and is given in Appendix 7.C. □

### 7.4.3 Concurrent Correctness

The concurrent subject reduction property expresses that each step of the concurrent evaluation of the expression can be mimicked by a number of steps in the concurrent evolution of its behavior.

Let us first relate the *configurations* $CI \;\&\; PP$ of the concurrent evaluation of expressions to the configurations $PB$ of the concurrent evolution of behaviors. We shall say that $CI \;\&\; PP$ is *cenv-related to* $PB$ if

$$\mathrm{Dom}(PP) = \mathrm{Dom}(PB) \text{ and } \mathrm{Dom}(cenv) = CI.$$

This ensures that we are dealing with the same process and channel identifiers. Furthermore, we say that $CI \;\&\; PP$ is *cenv-described by* $PB$ if it is *cenv*-related to $PB$ and if for all $pi \in \mathrm{Dom}(PP)$ there exists a type $t_{pi}$ such that

$$cenv \vdash PP\ pi : t_{pi} \;\&\; PB\ pi.$$

We shall also need to relate the *events ev* of the concurrent evaluation of expressions to the *actions a* of the concurrent evolution of behaviors. So assume that $CI \;\&\; PP$ is *cenv*-described by $PB$. Clearly we would expect $\text{FORK}_{\pi}\ pi$ to correspond to $\text{FORK}_{\pi}\ (PB\ pi)$ and $\text{CHAN}_l\ ci$ to correspond to $t$ CHAN $r$ when $cenv\ ci = t$ chan $r$ and $l \in r$; this is formalized by an auxiliary function denoted $\mathcal{A}\ (cenv, PB)$:

$$\begin{aligned}
\mathcal{A}\ (cenv, PB)\ \epsilon &= \epsilon, \\
\mathcal{A}\ (cenv, PB)\ (\text{CHAN}_l\ ci) &= t \text{ CHAN } r \quad \text{if } cenv\ ci = t \text{ chan } r, \\
&\qquad\qquad\quad \text{and } l \in r, \\
\mathcal{A}\ (cenv, PB)\ (\text{FORK}_{\pi}\ pi) &= \text{FORK}_{\pi}\ b \quad \text{if } PB\ pi = b, \\
\mathcal{A}\ (cenv, PB)\ (ci!, ci?) &= (r!t, r?t) \quad \text{if } cenv\ ci = t \text{ chan } r.
\end{aligned}$$

The final preparation is to introduce a notation for a sequence of steps in the concurrent evolution of behaviors. For this we write

$$PB \Longrightarrow^{\widehat{a}}_{ps} PB'$$

to mean that there exists $n \geq 0$ and configurations $PB_1, \cdots, PB_n$ such that

$$PB \Longrightarrow^{\epsilon}_{pi_1} \cdots \Longrightarrow^{\epsilon}_{pi_n} PB_n \Longrightarrow^{a}_{ps} PB',$$

where $pi_1, \cdots, pi_n$ are process identifiers from the list $ps$. Thus, the processes of $ps$ are allowed to perform some trivial actions before they engage in the joint action $a$. We then have the following result linking concurrent evaluation to concurrent evolution:

**Theorem 7.4.8.** Assume that

$$CI \;\&\; PP \longrightarrow^{ev}_{ps} CI' \;\&\; PP'$$

and that $CI \;\&\; PP$ is *cenv*-described by $PB$. Then there exists *cenv'* and $PB'$ such that

$$PB \Longrightarrow^{\widehat{a}}_{ps} PB',$$

where $CI' \;\&\; PP'$ is *cenv'*-described by $PB'$, $\mathcal{A}\ (cenv', PB')\ ev = a$, and we furthermore have $cenv' \lceil CI = cenv$.

**Proof.** The proof is by induction on the rules for concurrent evaluation and is given in Appendix 7.C. In this proof we exploit the soundness (Corollary 7.3.3) of the ordering $\sqsubseteq$ (Figure 7.2). However, it may be interesting to note that the laws **P1**, **P2**, **C1**, **C2**, **S1**, **E1**, **E2**, and **J1** suffice for carrying out the proof. □

## 7.5 Decidability Issues

We have previously established the soundness (Corollary 7.3.3) of the ordering $\sqsubseteq$ with respect to the simulation ordering $\underset{\sim}{\sqsubseteq}$; we also showed that we do not have completeness. It is a consequence of the results established in this section that *we do not desire completeness*. The reason is that $\underset{\sim}{\sqsubseteq}$ turns out to be undecidable whereas $\sqsubseteq$ (for proofs not involving **R1**) turns out to be decidable. We also discuss the consequences of incorporating the use of **R1** into our results.

### *7.5.1 Undecidability of the Simulation Ordering*

We now show that the simulation order $\underset{\sim}{\sqsubseteq}$ is undecidable by a reduction from the language containment problem for simple grammars [56]. This is in the spirit of the undecidability proof for the Basic Process Algebra [64] but due to the differences between our behaviors and the Basic Process Algebra it is simpler to perform a direct reduction.



Let $G = (V, T, P, S)$ be a simple grammar without useless nonterminals. To be specific, $V$ is a finite and nonempty set of nonterminals, $T$ is a finite and nonempty set of terminals, $P$ is a finite set of productions, and $S \in V$ is the start symbol. Each production of $P$ is of the form $A_0 \rightarrow a\ A_1 \ldots A_n$, where $A_i \in V$ and $a \in T$ and for any two productions $A_0 \rightarrow a\ A_1 \ldots A_n$ and $A_0 \rightarrow a\ A'_1 \ldots A'_{n'}$ we have $A_1 \ldots A_n = A'_1 \ldots A'_{n'}$. For each nonterminal $A$ there is a derivation from $S$ to a sentential form involving $A$ and from this to a terminal string. As a consequence every finite derivation starting from $S$ can be extended to one that ends in a terminal string.

As a first step we transform $G$ to $G' = (V', T', P', S')$, which operates over the symbols of our behaviors. Let $V'$ be a finite subset of behavior variables, $T'$ a finite set of behaviors of the form $\texttt{1!(unit} \rightarrow^{\epsilon})^n \texttt{unit}$, and let $\Theta$ be a bijection from $V$ to $V'$ and from $T$ to $T'$. Then $P'$ contains $\Theta(A \rightarrow a\ A_1 \ldots A_n)$ whenever $P$ contains $A \rightarrow a\ A_1 \ldots A_n$ and $S' = \Theta(S)$. Clearly $G'$ enjoys the same properties as $G$: it is simple, and all finite derivations from the start symbol to a sentential form may be extended to one that ends in a terminal string. Furthermore the language $\mathcal{L}_{G'}(S')$ generated by $G'$ is isomorphic to the language $\mathcal{L}_G(S)$ generated by $G$. Indeed, without loss of generality for the arguments that follow, one might assume $G = G'$.

As a second step we construct a behavior system $B = (b, \{\beta_i = b_i \mid i \leq k\})$ as follows: $b$ is $S'$, and if $\beta_i \in V'$ and $\beta_i \rightarrow a^1\ A^1_1 \ldots A^1_{n_1}, \cdots, \beta_i \rightarrow a^m A^m_1 \ldots A^m_{n_m}$ are all the $\beta_i$-productions in $P'$, we set

$$b_i = (a^1; A^1_1; \ldots; A^1_{n_1}) + \cdots + (a^m; A^m_1; \ldots; A^m_{n_m}).$$

By our assumptions we have $m > 0$ and each $n_j \geq 0$. We define the language $\mathcal{L}(B)$ generated by $B$ as

$$\mathcal{L}(b, \{\beta_i = b_i \mid i \leq k\}) = \{w \in T^* \mid b \overset{w}{\underset{B}{\Longrightarrow}}{}^* \surd\},$$

where $\underset{B}{\Longrightarrow}{}^*$ is the reflexive transitive closure of $\underset{B}{\Longrightarrow}$ and $\underset{B}{\Longrightarrow}$ is like $\Longrightarrow$ for behaviors but with the additional axiom $\beta_i \underset{B}{\Longrightarrow} b_i$. Clearly $\mathcal{L}(B') = \mathcal{L}_{G'}(S')$, and every finite derivation starting from $b$ can be extended to one ending in $\surd$. Also, each $b_i$ has all $a^1, \ldots, a^m$ to be distinct.

As a third step we perform a number of ministeps for reducing the number of equations. Each ministep transforms $(b_0, \{\beta_i = b_i \mid 0 < i \leq k\})$ to $(b'_0, \{\beta_i = b'_i \mid 0 < i < k\})$, where $b'_i = b_i[\beta \mapsto \text{REC}\beta_k.b_k]$. We thus end up with $B' = (b', \emptyset)$, where $b'$ has no free behavior variables. The system $B'$ enjoys the same properties as $B$ and has $\mathcal{L}(B') = \mathcal{L}(B)$ because these properties are preserved by all the ministeps.

Focusing the attention on $b'$ and the language $\mathcal{L}(b') = \{w \mid b' \overset{w}{\Longrightarrow}{}^* \surd\}$ generated by it we have $\mathcal{L}(b') = \mathcal{L}(B')$. Also, each finite derivation from $b'$ may be extended to one ending in $\surd$; we write $\models_N b'$ for this, and it is a consequence that $b'$ has no free behavior variables. Note that if $b' \overset{w}{\Longrightarrow}{}^* b''$

then $\models_N b'$ implies $\models_N b''$. Furthermore, each sum of behaviors in $b'$ must have distinct first actions; we record this by decreeing $\vdash_D b'$, where $\vdash_D$ is defined inductively by

$$\vdash_D \epsilon \qquad \vdash_D 1 \,!\, t \qquad \vdash_D \beta$$

$$\frac{\vdash_D b_1 \quad \vdash_D b_2}{\vdash_D b_1; b_2} \qquad \frac{\vdash_D b}{\vdash_D \text{REC}\beta.b}$$

$$\frac{\vdash_D b_1 \ldots \vdash_D b_n}{\vdash_D (p_1; b_1) + \ldots + (p_n; b_n)} \quad (p_1, \ldots, p_n) \text{ simple}$$

where $(p_1, \ldots, p_n)$ is "simple" if $n > 0$ and each $p_i$ is a primitive action different from $\epsilon$ and all $p_i$'s are distinct. (Note that we did not define $\vdash_D 1 \,?\, t, \vdash_D \text{FORK}_\pi(b)$, or $\vdash_D t \text{ CHAN } r$ because these constructs cannot appear in $b'$.)

We now need two technical results:

**Lemma 7.5.1.** *(i)* If $\vdash_D b_1, \models_N b_1$ and $b_1 \Rightarrow^\epsilon b_2$ then $\vdash_D b_2, \models_N b_2$ and $\mathcal{L}(b_2) \subseteq \mathcal{L}(b_1)$; furthermore, if $pw \in \mathcal{L}(b_2)$ for some $p \neq \epsilon$ then $\{pw \mid pw \in \mathcal{L}(b_1)\} = \{pw \mid pw \in \mathcal{L}(b_2)\}$.

*(ii)* If $\vdash_D b_1, \models_N b_1$ and $b_1 \Rightarrow^{\widehat{p}} b_2$ with $p \neq \epsilon$ then, $\vdash_D b_2, \models_N b_2$ and $\mathcal{L}(b_2) = \{w \mid pw \in \mathcal{L}(b_1)\}$.

**Proof.** We already have $\models_N b_2$ from a previous observation. The remaining claims are proved by induction on $\vdash_D b_1$. We omit the details. □

**Lemma 7.5.2.** If $\vdash_D b', \vdash_D b'', \models_N b', \models_N b''$ then $\mathcal{L}(b') \subseteq \mathcal{L}(b'') \Leftrightarrow b' \sqsubseteq\!\!\!\sim b''$.

**Proof.** See Appendix 7.D. □

We can now state our main result:

**Proposition 7.5.3.** $\sqsubseteq\!\!\!\sim$ is undecidable.

**Proof.** Language containment for simple grammars with no useless nonterminals is undecidable due to [56]. The above procedures transform simple grammars $G_1$ and $G_2$ with no useless nonterminals into behaviors $b'_1$ and $b'_2$ such that $\mathcal{L}_{G_1}(S_1) \subseteq \mathcal{L}_{G_2}(S_2)$ amounts to $\mathcal{L}(b'_1) \subseteq \mathcal{L}(b'_2)$ and where $\vdash_D b'_1, \vdash_D b'_2, \models_N b'_1$ and $\models_N b'_2$. By Lemma 7.5.2 $\mathcal{L}_{G_1}(S_1) \subseteq \mathcal{L}_{G_2}(S_2)$ is then equivalent to $b_1 \sqsubseteq\!\!\!\sim b_2$. If the latter were to be decidable we would have a contradiction. □

### 7.5.2 *Decidability of the Syntactic Ordering*

Although $\sqsubseteq\!\!\!\sim$ is undecidable we can show that the subset of the ordering $\sqsubseteq$ defined by excluding the axiom **R1** from the laws of Figure 7.2 is indeed



$$
\begin{array}{lcl}
[\![\epsilon]\!] & = & \epsilon \\
[\![r!t]\!] & = & r!\lceil t \rceil \\
[\![r?t]\!] & = & r?\lceil t \rceil \\
[\![t \text{ CHAN } r]\!] & = & \lceil t \rceil \text{ CHAN } r \\
[\![\beta]\!] & = & \beta \\
[\![\text{FORK}_\pi\ b]\!] & = & \text{FORK}_\pi\ [\![b]\!] \\
[\![b_1; b_2]\!] & = & \text{seq}([\![b_1]\!], [\![b_2]\!]) \\
[\![b_1 + b_2]\!] & = & [\![b_1]\!] + [\![b_2]\!] \\
[\![\text{REC}\beta.\ b]\!] & = & \text{REC}\beta.[\![b]\!] \\
\text{seq}(bs, bn_0) & = & \begin{cases} bs & \text{if } bn_0 = \epsilon \\ bn_0 & \text{if } bs = \epsilon \wedge bn_0 \neq \epsilon \\ bp; bn_0 & \text{if } bs = bp \wedge bn_0 \neq \epsilon \\ bp; \text{seq}(bn, bn_0) & \text{if } bs = bp; bn \wedge bn_0 \neq \epsilon \end{cases} \\
\text{seq}(bn + bn', bn_0) & = & \text{seq}(bn, bn_0) + \text{seq}(bn', bn_0)
\end{array}
$$

FIGURE 7.11. Transformation to canonical form

decidable. More precisely, we shall give an algorithm that given two behaviors $b_1$ and $b_2$ will decide whether $b_1 \sqsubseteq b_2$ without using **R1** and we shall show that it is sound and complete.

The algorithm proceeds in two stages. First it transforms the behaviors into canonical forms and then it checks the ordering between the canonical forms. A behavior is in *canonical form* if it is constructed according to the nonterminal $bn$ of the following grammar:

$$bn ::= bs \mid bn + bn$$

$$bs ::= \epsilon \mid bp \mid bp; bn$$

$$bp ::= r!t \mid r?t \mid t \text{ CHAN } r \mid \beta \mid \text{FORK}_\pi\ bn \mid \text{REC}\beta.bn$$

This means that a canonical behavior is a sum of behaviors. A summand is either $\epsilon$ or it begins with a primitive behavior and is possibly followed by a canonical behavior. A primitive behavior is one of $r!t$, $r?t$, $t$ CHAN $r$, $\beta$, $\text{FORK}_\pi\ bn$ and $\text{REC}\beta.bn$, where the last two have bodies in canonical form.

Each behavior $b$ can be transformed into an equivalent canonical behavior $[\![b]\!]$. The translation is specified in Figure 7.11 and proceeds according to the structure of the behavior. It uses the auxiliary operation $\lceil \cdots \rceil$ that will transform a type into a form where bound behavior variables are $\alpha$-renamed in the manner of deBruin indices. So for example $\texttt{int} \rightarrow^{\text{REC}\beta_1.\beta_1}$

$$\frac{bs \text{ ord}_s bs_0}{bs \text{ ord } bs_0} \qquad \frac{bs \text{ ord } bn_0}{bs \text{ ord } bn_0 + bn_0'}$$

$$\frac{bs \text{ ord } bn_0'}{bs \text{ ord } bn_0 + bn_0'} \qquad \frac{bn \text{ ord } bn_0,\ bn' \text{ ord } bn_0}{bn + bn' \text{ ord } bn_0}$$

$$\epsilon \text{ ord}_s \epsilon \qquad \frac{bp \text{ ord}_p bp_0}{bp \text{ ord}_s bp_0}$$

$$\frac{bp \text{ ord } bp_0 \quad bn \text{ ord } bn_0}{bp; bn \text{ ord}_s bp_0; bn_0}$$

$$r!t \text{ ord}_p r!t \qquad r?t \text{ ord}_p r?t$$

$$t \text{ CHAN } r \text{ ord}_p t \text{ CHAN } r \qquad \beta \text{ ord}_p \beta$$

$$\frac{bn \text{ ord } bn_0}{\text{FORK}_\pi\ bn \text{ ord}_p \text{ FORK}_\pi\ bn_0}$$

$$\frac{bn[\beta \mapsto \beta'] \text{ ord } bn_0[\beta_0 \mapsto \beta']}{\text{REC}\beta.\ bn \text{ ord}_p \text{ REC}\beta_0.\ bn_0}$$

if $\beta' \notin FV(\text{REC}\beta.\ bn) \cup FV(\text{REC}\beta_0.\ bn_0)$

FIGURE 7.12. Checking the ordering for behaviors in canonical form

int and int $\rightarrow^{\text{REC}\beta_2.\beta_2}$ int will be transformed into the same type int $\rightarrow^{\text{REC}\beta'.\beta'}$ int for some unique $\beta'$. Furthermore, $[\![\cdots]\!]$ uses the auxiliary function seq that takes two behaviors $bn_1$ and $bn_2$ in canonical form and constructs a behavior equivalent to $bn_1; bn_2$ but in canonical form. Basically, the translation applies the laws **S1**, **S2**, **E1**, and **E2** to ensure that behaviors are written in a certain form. The translation could easily be extended to remove duplicate summands.

**Lemma 7.5.4.** For all $b$: $[\![b]\!]$ is in canonical form and $[\![b]\!] \equiv b$.

**Proof.** It is easy to verify that $[\![b]\!]$ is in canonical form. The equivalence of $b$ and $[\![b]\!]$ is proved by a straightforward structural induction on $b$ and uses

$$\text{seq}(bn, bn_0) \equiv bn; bn_0,$$

which can easily be proved by structural induction on $bn$. □

Given two behaviors $bn_1$ and $bn_2$ in canonical form, we can now present a method for deciding whether $bn_1 \sqsubseteq bn_2$ holds. It is presented in Figure 7.12 in the form of an inference system that axiomatizes the three relations ord, $\text{ord}_s$, and $\text{ord}_p$. We shall see that $bn_1$ ord $bn_2$ amounts to $bn_1 \sqsubseteq bn_2$, that



$bs_1$ $\text{ord}_s$ $bs_2$ amounts to $bs_1 \sqsubseteq bs_2$, and that $bp_1$ $\text{ord}_p$ $bp_2$ amounts to $bp_1 \sqsubseteq bp_2$.

Intuitively it should be clear that the inference system may be converted into the definition of three terminating functions ord, $\text{ord}_s$, and $\text{ord}_p$ defined by pattern matching upon their arguments and with a catch-all case giving *false*. The only case that is slightly nontrivial is

$$\text{ord}(bs, bn_0 + bn_0') = \text{ord}(bs, bn_0) \vee \text{ord}(bs, bn_0') \vee \mathit{false},$$

since it is the only case where more than one rule may be appropriate.

Formally, we verify our claim that Figure 7.12 defines an algorithm by

**Lemma 7.5.5.** The relations ord, $\text{ord}_s$, and $\text{ord}_p$ are decidable.

**Proof.** Define $size(b)$ to be the size (say length of presentation) of the behavior $b$. Define the measure $\mu$ by

$$\mu(bn_1 \text{ ord } bn_2) = 2 + 3(size(bn_1) + size(bn_2)),$$
$$\mu(bs_1 \text{ ord}_s bs_2) = 1 + 3(size(bs_1) + size(bs_2)),$$
$$\mu(bp_1 \text{ ord}_p bp_2) = 0 + 3(size(bp_1) + size(bp_2)),$$

and note that $\mu$ always produces a nonnegative integer. Next inspect all rules of Figure 7.12 and note that the measure $\mu$ of the conclusion is always strictly larger than the maximum of the measure $\mu$ of the premises. This shows that an inference tree for a statement has height at most the measure $\mu$ of that statement. Since there are finitely many binary trees of a given height it would suffice to search each one in turn. □

Soundness of the method is expressed by

**Theorem 7.5.6.** If $[\![b]\!]$ ord $[\![b_0]\!]$ then $b \sqsubseteq b_0$; furthermore, the inference of $b \sqsubseteq b_0$ need not use **R1**.

**Proof.** Using Lemma 7.5.4 it suffices to prove the (stronger) result

$$\text{if } bn \text{ ord } bn_0 \text{ then } bn \sqsubseteq bn_0, \tag{1}$$
$$\text{if } bs \text{ ord}_s bs_0 \text{ then } bs \sqsubseteq bs_0, \tag{2}$$
$$\text{if } bp \text{ ord}_p bp_0 \text{ then } bp \sqsubseteq bp_0. \tag{3}$$

The proof is by a straightforward induction on the inference tree. □

To prove completeness of the algorithm we shall first give an alternative formulation of when $bn$ ord $bn_0$. First we define $\Sigma(bn)$ as the *set of summands* of $bn$:

$$\Sigma(bs) = \{bs\},$$
$$\Sigma(bn + bn') = \Sigma(bn) \cup \Sigma(bn').$$

Then we have (reminiscent of the Egli-Milner ordering)

**Lemma 7.5.7.** $bn_1$ ord $bn_2$ if and only if for all $bs_1 \in \Sigma(bn_1)$ there exists $bs_2 \in \Sigma(bn_2)$ such that $bs_1$ $\mathrm{ord}_s$ $bs_2$.

**Proof.** The proof of "only if" is by a straightforward induction on the inference tree of $bn_1$ ord $bn_2$ (tracing the lower part involving ord only). The proof of "if" amounts to a straightforward construction of an inference tree. □

**Theorem 7.5.8.** If one can infer $b \sqsubseteq b_0$ without using **R1**, then $[\![b]\!]$ ord $[\![b_0]\!]$.

**Proof.** We prove that $[\![\cdots]\!]$ ord $[\![\cdots]\!]$ is a model for $\cdots \sqsubseteq \cdots$. Consult Appendix 7.D for the details. □

To see that we do *not* have a counterpart of the law **R1** consider the canonical behavior $\mathrm{REC}\beta.(r!\mathtt{int};\beta+\epsilon)$. Here

$$\Sigma(\mathrm{REC}\beta.(r!\mathtt{int};\beta+\epsilon)) = \{\mathrm{REC}\beta.(r!\mathtt{int};\beta+\epsilon)\},$$

$$\Sigma(r!\mathtt{int};\mathrm{REC}\beta.(r!\mathtt{int};\beta+\epsilon)+\epsilon) = \{r!\mathtt{int};\mathrm{REC}\beta.(r!\mathtt{int};\beta+\epsilon),\epsilon\},$$

and it is easy to see that neither

$$\mathrm{REC}\beta.(r!\mathtt{int};\beta+\epsilon)\ \mathrm{ord}_s\ r!\mathtt{int};\mathrm{REC}\beta.(r!\mathtt{int};\beta+\epsilon),$$

nor

$$\mathrm{REC}\beta.(r!\mathtt{int};\beta+\epsilon)\ \mathrm{ord}_s\ \epsilon.$$

*Towards a Larger and Still Decidable Ordering*

There are two approaches that may be pursued in order to extend our algorithm for deciding $\sqsubseteq$ to allow also the use of **R1**. One is to concentrate on **R1** and to build further knowledge into the algorithm. The caveat of this approach is that one risks non-termination due to the unbounded number of times that **R1** might be used. To overcome this one might try to apply automata-based techniques to discover when we repeat the same "pattern of recursion." This leads (we believe) to the second approach, where we must extend Figure 7.2 with new axioms and rules in order to ensure the soundness of the automata-based techniques: in particular, rules that allow some kind of induction to be performed. An example (perhaps derived) rule might allow one to deduce $\mathrm{REC}\beta.b_0 \sqsubseteq b$ from $b_0[\beta \mapsto b] \sqsubseteq b$ (under suitable side conditions).

This then raises an important question. If we decide to extend Figure 7.2, does this invalidate the other results we have established? Let us define *an admissible ordering* to be an inductively defined ordering between behaviors such that *(i)* it is a simulation, *(ii)* all axioms and rules maintain the



statements of Facts 7.2.5 and 7.2.6, and *(iii)* it contains all axioms and rules of Figure 7.2 (possibly omitting **C4** and **R1**). We believe that all results (except of course decidability of the ordering) would continue to hold if $\sqsubseteq$ of Figure 7.2 were replaced by another ordering provided it is *admissible*.

At present it is unclear which is the better route to pursue in order to obtain a decidable ordering that is still sound with respect to the simulation ordering

## 7.6 Conclusion

The starting point of this work has been an existing programming language, namely CML. We have developed a type and behavior inference system for a subset of this language: as usual the *types* talk about the sets of *values* upon which a program operates whereas the *behaviors* talk about the *communications* (or more generally the computations) that take place when the program executes. This is formally expressed by the subject reduction property that states that the execution of the CML program is mimicked by the evolution of its behavior viewed as a term in a process algebra.

Previous work [151, 152] shows how Standard ML's type system can be extended to take care of the concurrency constructs of CML; however, the polymorphic `let`-construct is handled in a rather restricted way by distinguishing between expansive and nonexpansive `let`-constructs. This is overcome by our type and behavior system: the behaviors contain information that allows us to handle polymorphism properly much as in the type and effect systems developed for type inference of Standard ML with references [160].

The structure of behaviors is much more refined than the sets of [160], as they also express the *causality* of the various communications that take place during computation. Because of this the behaviors can be used as a basis for a number of interesting program analyses of the communication structure of programs. Let us illustrate this with a few examples.

*Analysis for finite communication topology*

A CML program may create unboundedly many processes and channels. Since hardware is finite, an implementation may have to map many processes and channels to a "smaller" finite architecture. This task becomes easier if the number of processes and channels created in the program do not exceed the resources (processors and links) that are available at the hardware level. Furthermore, a much smaller version of the run-time system will be needed: there is no need for multitasking and multiplexing. In [125] we develop an analysis of behaviors that detects whether a CML program has a finite topology. As an example consider the behavior

$$\text{FORK}_\pi\ (\text{REC}\ \beta'.\ ((\rho_1?\alpha_1;\ b;\ \rho_2!\alpha_2;\ \beta') + \rho_0!\text{unit}))$$

associated with the node function in Example 7.2.2. Here the analysis will determine that if $b$ does not create any processes and channels then the node function will at most create one process and no channels; thus it has a finite topology. On the other hand, the pipe function has behavior

$$\begin{array}{ll}\text{REC}\ \beta'. & (\text{FORK}_\pi\ (\text{REC}\ \beta''.\ (((\rho_1 + l)?\alpha;\ \rho_2!\alpha;\ \beta'') + \rho_0!\text{unit})) \\ + & (\alpha\ \text{CHAN}\ (\rho_1 + l); \\ & \text{FORK}_\pi\ (\text{REC}\ \beta''.\ (((\rho_1 + l)?\alpha;\ b;\ \rho_2!\alpha;\ \beta'') + \rho_0!\text{unit})); \\ & \beta')).\end{array}$$

The analysis of [125] will deduce that any number of processes and any number of channels may be created so the pipe function does not have a finite topology.

*Processor allocation*

Often it will be the case that the number of processors of the hardware is less than the number of processes created by the CML program. Thus one has to resort to multitasking and decide how to allocate the various processes on the available processors. Basically there are two approaches: in *static processor allocation* it is decided at compile-time where all instances of a given process should reside at run-time, whereas in *dynamic processor allocation* it is decided at run-time. In both cases it is useful to have information about the requirements of the processes; for example, we might like to know which channels are needed for communication and how many times. In [126] we present an analysis showing how such information can be obtained from the behaviors. We illustrate this for the pipe function.

In the case of static processor allocation we will decide that all processes corresponding to the occurrence of $\texttt{fork}_\pi$ in the CML program will reside on the same processor. Thus the requirements of that processor are obtained by *accumulating* the requirements of *all* the processes with label $\pi$. So for the pipe function we see that there will be many inputs over channels in $\rho_1 + l$, many outputs over channels in $\rho_2$, and *many* outputs over channels in $\rho_0$. The latter result may be surprising because each of the processes labeled $\pi$ will communicate at most once over $\rho_0$. However, there may be many processes labeled $\pi$, and since they all will reside on the same processor this processor must be prepared to do many communications over $\rho_0$. The situation is different for dynamic processor allocation. Here we do not accumulate the requirements of each process with a specific label; instead we estimate the *maximal* requirements of all instances of the process, and for the pipe function we get that each process labeled $\pi$ will communicate over $\rho_0$ at most once; the results for $\rho_1 + l$ and $\rho_2$ are as for static processor allocation.



*Acknowledgment*

Discussions with Torben Amtoft, Fritz Henglein, Pierre Jouvelot, Jean-Pierre Talpin, Bent Thomsen, and Mads Tofte have been most stimulating.

# Appendix 7.A Syntactic Properties of the Typing System

**Proof of Lemma 7.2.8.** We need a somewhat stronger induction hypothesis. Define $tenv' \succeq tenv$ to mean that $\text{Dom}(tenv') = \text{Dom}(tenv)$ and that $tenv'\,x \succeq tenv\,x$ for all $x \in \text{Dom}(tenv)$. We then claim that

$$\text{if } tenv \vdash e : t\ \&\ b \text{ and } tenv' \succeq tenv \text{ then } tenv' \vdash e : t\ \&\ b.$$

The proof is by induction on the structure of the inference of $tenv \vdash e : t\ \&\ b$. In the case of let-polymorphism the key observation is that $ts' \succeq ts$ implies $FV(ts') \subseteq FV(ts)$. To see that this is indeed the case let $ts \succ t$ be chosen such that $FV(ts) = FV(t)$, for example by mapping all quantified variables of $ts$ to ground terms; since $ts' \succ t$ we immediately have $FV(ts') \subseteq FV(t)$ and this establishes the result. Using this observation it is then immediate that $\text{gen}(tenv', b)t \succeq \text{gen}(tenv, b)t$. □

**Proof of Lemma 7.2.10.** This is a structural induction on $e$ using Fact 7.2.6. Only the case of let-polymorphism is non-trivial. In this case assume

$$tenv \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t\ \&\ b.$$

Then $b_1; b_2 \sqsubseteq b$ and

$$\begin{array}{l}tenv \vdash e_1 : t_1\ \&\ b_1, \\ tenv[x \mapsto ts] \vdash e_2 : t\ \&\ b_2,\end{array}$$

where $ts = \text{gen}(tenv, b_1)\ t_1$. Let $\{\vec{\alpha}\vec{\beta}\vec{\rho}\} = FV(t_1)\backslash(FV(tenv) \cup FV(b_1))$ and let $\mu$ be a renaming of $\vec{\alpha}\vec{\beta}\vec{\rho}$ to fresh and distinct variables $\vec{\alpha_0}\,\vec{\beta_0}\,\vec{\rho_0}$. Then the induction hypothesis gives

$$\begin{array}{l}\theta\,(\mu\ tenv) \vdash e_1 : \theta\,(\mu\ t_1)\ \&\ \theta\,(\mu\ b_1), \\ (\theta\ tenv)[x \mapsto \theta\ ts] \vdash e_2 : \theta\ t\ \&\ \theta\ b_2,\end{array}$$

and the former can be rewritten as

$$\theta\ tenv \vdash e_1 : \theta\,(\mu\ t_1)\ \&\ \theta\ b_1.$$

Let $ts' = \text{gen}(\theta\ tenv, \theta\ b_1)(\theta\,(\mu\ t_1))$. One can prove that $ts' \succeq \theta\ ts$, and Lemma 7.2.8 gives

$$(\theta\ tenv)[x \mapsto ts'] \vdash e_2 : \theta\ t\ \&\ \theta\ b_2.$$

From Fact 7.2.6 we get $\theta\, b_1; \theta\, b_2 \sqsubseteq \theta\, b$, and so using the rule for let-polymorphism we get the required

$$\theta\, tenv \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \theta\, t\ \&\ \theta\, b.$$

This completes the proof. □

**Proof of Lemma 7.2.12.** We proceed by induction on the structure of the inference $tenv \vdash e : t\,\&\,b$. Due to the assumption that $tenv'$ redefines no identifier of $tenv$ all cases except let-polymorphism are straightforward; in particular the cases for variables and abstraction. For the case of let-polymorphism it suffices by Lemma 7.2.8 to show

$$\mathsf{gen}(tenv\, tenv', b_1)t_1 \ \succeq\ \mathsf{gen}(tenv, b_1)t_1,$$

for which it suffices to show

$$FV(t_1)\backslash(FV(b_1)\cup FV(tenv)\cup FV(tenv')) \supseteq FV(t_1)\backslash(FV(b_1)\cup FV(tenv)).$$

For this to be false we must have $\gamma \in FV(t_1), \gamma \in FV(tenv'), \gamma \notin FV(b_1)$, and $\gamma \notin FV(tenv)$; but then $\gamma \notin IV(tenv \vdash e : t\,\&\,b)$ and so $\gamma \notin FV(t_1)$, and we have the desired contradiction. □

**Proof of Corollary 7.2.13.** Let $X = FV(tenv') \setminus FV(tenv)$. Define the substitution $\theta$ such that $\mathrm{Dom}(\theta) = X$, such that $\{\gamma_1, \gamma_2\} \subseteq X \wedge \theta\gamma_1 = \theta\gamma_2 \Rightarrow \gamma_1 = \gamma_2$, and such that $\{\theta\gamma \mid \gamma \in X\}$ is disjoint with all variables occurring in the proof of $tenv \vdash e : t\,\&\,b$. Then $FV(\theta(tenv')) \cap IV(tenv \vdash e : t\,\&\,b) = \emptyset$. It follows from Lemma 7.2.12 that $tenv\,(\theta(tenv')) \vdash e : t\,\&\,b$. Next define the substitution $\theta'$ such that $\mathrm{Dom}(\theta') = \{\theta\gamma \mid \gamma \in X\}$ and $\theta'(\theta\gamma) = \gamma$ for all $\gamma \in X$. It follows from Lemma 7.2.10 that $\theta'(tenv)\,\theta'(\theta(tenv')) \vdash e : \theta'(t)\,\&\,\theta'(b)$. Clearly $\theta'(tenv) = tenv$, $\theta'(t) = t$ and $\theta'(b) = b$ by the assumptions on $\{\theta\gamma \mid \gamma \in X\}$; furthermore $\theta'(\theta(tenv')) = tenv'$. This proves the result. □

**Proof of Lemma 7.2.14.** From the assumption and Lemma 7.2.8 we have

$$tenv[x \mapsto \mathsf{gen}(tenv, \epsilon)t_0] \vdash e : t\ \&\ b \qquad (1)$$

We shall now assume that the variables of (1) not occurring in $\mathsf{gen}(tenv, \epsilon)t_0$ are disjoint with all variables of

$$tenv \vdash e_0 : t_0\ \&\ \epsilon \qquad (2)$$

except those in $FV(tenv)$; this is merely an application of Fact 7.2.11. Furthermore we assume that all defined identifiers of $e$ have been alpha-renamed so as not to occur in $tenv$; this ensures that no alpha-renaming is needed when performing the substitution $e[e_0/x]$.

We now want to modify (1) by replacing each node



$$tenv[x \mapsto \mathsf{gen}(tenv, \epsilon)t_0]tenv' \vdash e' : t'\ \&\ b'$$

by

$$tenv\, tenv' \vdash e' : t'\ \&\ b' \qquad \text{if } e' \neq x \text{ or } x \in \mathrm{Dom}(tenv'),$$
$$tenv\, tenv' \vdash e_0 : t'\ \&\ b' \qquad \text{if } e' = x \text{ and } x \notin \mathrm{Dom}(tenv').$$

The first part of this transformation is immediate, whereas the second part needs to be obtained from (2). First note that

$$\mathsf{gen}(tenv, \epsilon)t_0 \succ t' \quad \text{and} \quad \epsilon \sqsubseteq b'$$

and that $\theta\, t_0 = t'$ for some substitution $\theta$ defined on $FV(t_0) \setminus FV(tenv)$. Using Lemma 7.2.12 we get

$$tenv\, tenv' \vdash e_0 : t_0\ \&\ \epsilon,$$

where $\mathrm{Dom}(tenv') \cap \mathrm{Dom}(tenv) = \emptyset$ follows from the alpha-renaming on $e$, and $FV(tenv') \cap IV(tenv \vdash e_0 : t_0\,\&\,\epsilon) = \emptyset$ follows from the assumption that all variables of (1) not occurring in $\mathsf{gen}(tenv, \epsilon)t_0$ are disjoint with those of (2) except for those in $FV(tenv)$. We now use Lemma 7.2.10 and Fact 7.2.9 to get

$$tenv\, tenv' \vdash e_0 : t'\ \&\ b'.$$

This shows the well-definedness of the transformation.

The final step is to prove by structural induction that the constructed structure is indeed a proof of

$$tenv \vdash e[e_0/x] : t\ \&\ b.$$

This is a straightforward induction on the proof tree for (1). In the case of let-polymorphism note that

$$FV(tenv) = FV(tenv[x \mapsto \mathsf{gen}(tenv, \epsilon)t_0])$$

because $FV(\mathsf{gen}(tenv, \epsilon)t_0) = FV(t_0) \cap FV(tenv)$. □

## Appendix 7.B Semantic Properties of the Ordering

**Proof of Proposition 7.3.2.** We show that the laws of Figure 7.2 fulfill the requirements, that is, whenever $b_1$ and $b_2$ are closed, $b_1 \sqsubseteq b_2$ and

$$b_1 \Rightarrow^{p_1} \flat_1,$$

then there exists $p_2$ and $\flat_2$ such that

$$b_2 \Rightarrow^{\widehat{p_2}} \flat_2,\ p_1 \sqsubseteq^{\partial} p_2 \text{ and } \flat_1 \sqsubseteq \flat_2.$$

For open behaviors the result follows from Fact 7.2.6.

**The case P1** is immediate.

**The case P2.** Then $b_1 \sqsubseteq b_3$ because $b_1 \sqsubseteq b_2$ and $b_2 \sqsubseteq b_3$. From $b_1 \Rightarrow^{p_1} \flat_1$ and the induction hypothesis (IH) we get

$$\begin{array}{l} b_2 \Rightarrow^{\widehat{p_2}} \flat_2, \\ p_1 \sqsubseteq^{\partial} p_2 \text{ and } \flat_1 \sqsubseteq \flat_2, \end{array} \tag{1}$$

for some $p_2$ and $\flat_2$. By induction on the length of the derivation sequence of (1) we will show the required

$$\begin{array}{l} b_3 \Rightarrow^{\widehat{p_3}} \flat_3, \\ p_1 \sqsubseteq^{\partial} p_3 \text{ and } \flat_1 \sqsubseteq \flat_3. \end{array} \tag{2}$$

If the length of (1) is 1 then the induction hypothesis gives

$$\begin{array}{l} b_3 \Rightarrow^{\widehat{p_3}} \flat_3, \\ p_2 \sqsubseteq^{\partial} p_3 \text{ and } \flat_2 \sqsubseteq \flat_3, \end{array}$$

and (2) follows using **P2**. For the induction step assume the length of (1) is $n+1$. Then

$$b_2 \Rightarrow^{\epsilon} b_2' \Rightarrow^{\widehat{p_2}} \flat_2,$$

and IH applied to the first step gives

$$\begin{array}{l} b_3 \Rightarrow^{\widehat{p_3'}} \flat_3', \\ \epsilon \sqsubseteq^{\partial} p_3' \text{ and } b_2' \sqsubseteq \flat_3'. \end{array}$$

Now $p_3' = \epsilon$ is the only possibility and also $\flat_3' \neq \surd$ must be the the case. Next, IH applied to $b_2' \Rightarrow^{\widehat{p_2}} \flat_2$ with $b_2' \sqsubseteq \flat_3'$ gives

$$\begin{array}{l} \flat_3' \Rightarrow^{\widehat{p_3}} \flat_3, \\ p_1 \sqsubseteq^{\partial} p_3 \text{ and } \flat_1 \sqsubseteq \flat_3. \end{array}$$

Thus we have $b_3 \Rightarrow^{\widehat{p_3}} \flat_3$ and the result follows.

**The case C1.** Assume $b_1; b_3 \sqsubseteq b_2; b_4$ because $b_1 \sqsubseteq b_2$ and $b_3 \sqsubseteq b_4$. Furthermore assume

$$b_1;\ b_2 \Rightarrow^{p_1} \flat_1.$$

There are two interesting subcases:

$$\begin{array}{lll} (i) & b_1 \Rightarrow^{p_1} b_1' & \text{and} \quad \flat_1 = b_1'; b_2, \\ (ii) & b_1 \Rightarrow^{p_1} \surd & \text{and} \quad \flat_1 = b_2. \end{array}$$

In subcase *(i)* IH gives $b_2 \Rightarrow^{\widehat{p_2}} \flat_2'$, $p_1 \sqsubseteq^{\partial} p_2$, and $b_1' \sqsubseteq \flat_2'$. Since $b_1' \neq \surd$ we have $\flat_2' \neq \surd$; so

$$b_2; b_4 \Rightarrow^{\widehat{p_2}} \flat_2'; b_4.$$



Using **P1**, **P2**, and **C1** we get $\flat_1 \sqsubseteq \flat_2'; b_4$ as required. In subcase *(ii)* IH gives $b_2 \Rightarrow^{\widehat{p_2}} \flat_2'$, $p_1 \sqsubseteq^{\partial} p_2$, and $\surd \sqsubseteq \flat_2'$. Thus $\flat_2' = \surd$ must be the case and

$$b_2;\ b_4 \Rightarrow^{\widehat{p_2}} b_4.$$

The result then follows.

**The case C2.** Assume $b_1 + b_3 \sqsubseteq b_2 + b_4$ because $b_1 \sqsubseteq b_2$ and $b_3 \sqsubseteq b_4$. Furthermore assume that

$$b_1 + b_3 \Rightarrow^{p_1} \flat_1.$$

The only interesting case is when $b_i \Rightarrow^{p_1} \flat_1$ for $i = 1$ or $i = 3$. Then IH gives $b_{i+1} \Rightarrow^{\widehat{p_2}} \flat_2$ for $p_1 \sqsubseteq^{\partial} p_2$ and $\flat_1 \sqsubseteq \flat_2$. So we get the required

$$b_2 + b_4 \Rightarrow^{\widehat{p_2}} \flat_2.$$

**The case C3.** Assume $\text{FORK}_\pi\ b_1 \sqsubseteq \text{FORK}_\pi\ b_2$ because $b_1 \sqsubseteq b_2$. Furthermore, assume

$$\text{FORK}_\pi\ b_1 \Rightarrow^{p_1} \flat_1.$$

The only interesting case is when $p_1 = \text{FORK}_\pi\ b_1$ and $\flat_1 = \epsilon$. Clearly $\text{FORK}_\pi\ b_2 \Rightarrow^{\widehat{p_2}} \epsilon$ for $p_2 = \text{FORK}_\pi\ b_2$, and since $\text{FORK}_\pi\ b_1 \sqsubseteq^{\partial} \text{FORK}_\pi\ b_2$ and $\epsilon \sqsubseteq \epsilon$ the result follows.

**The case C4.** Assume $\text{REC}\beta.b_1 \sqsubseteq \text{REC}\beta.b_2$ because $b_1 \sqsubseteq b_2$. Furthermore, assume

$$\text{REC}\beta.b_1 \Rightarrow^{p_1} \flat_1.$$

The only interesting case is when $p_1 = \epsilon$ and $\flat_1 = b_1[\beta \mapsto \text{REC}\beta.b_1]$. Clearly

$$\text{REC}\beta.b_2 \Rightarrow^{\widehat{\epsilon}} b_2[\beta \mapsto \text{REC}\beta.b_2].$$

We have

$$\begin{array}{ll} b_1[\beta \mapsto \text{REC}\beta.b_1] & \sqsubseteq \text{REC}\beta.b_1 \\ & \sqsubseteq \text{REC}\beta.b_2 \\ & \sqsubseteq b_2[\beta \mapsto \text{REC}\beta.b_2] \end{array}$$

using **P2**, **R1**, and the assumption. Thus the result follows.

**The case S1.** First we study $b_1; (b_2; b_3) \sqsubseteq (b_1; b_2); b_3$. So assume

$$b_1; (b_2; b_3) \Rightarrow^{p} \flat.$$

There are two interesting cases:

$$\begin{array}{lll} (i) & b_1 \Rightarrow^{p} b_1' & \text{and} \quad \flat = b_1'; (b_2; b_3), \\ (ii) & b_1 \Rightarrow^{p} \surd & \text{and} \quad \flat = b_2; b_3. \end{array}$$

In subcase *(i)* we get $b_1; b_2 \Rightarrow^{p} b_1'; b_2$ and then $(b_1; b_2); b_3 \Rightarrow^{p} (b_1'; b_2); b_3$. Since $p \sqsubseteq^{\partial} p$ and $b_1'; (b_2; b_3) \sqsubseteq (b_1'; b_2); b_3$ (follows from **S1**) we get the

required result. In subcase *(ii)* we have $b_1; b_2 \Rightarrow^p b_2$ and then $(b_1; b_2); b_3 \Rightarrow^p b_2; b_3$. The result follows using **P1**.

Next we study $(b_1; b_2); b_3 \sqsubseteq b_1; (b_2; b_3)$. So assume

$$(b_1; b_2); b_3 \Rightarrow^p \flat.$$

It cannot be the case that $b_1; b_2 \Rightarrow^p \surd$; so the only interesting case is when

$$b_1; b_2 \Rightarrow^p b \text{ and } \flat = b; b_3.$$

We have the two subcases

$$\begin{array}{llll} (i) & b_1 \Rightarrow^p b_1' & \text{and} & b = b_1'; b_2, \\ (ii) & b_1 \Rightarrow^p \surd & \text{and} & b = b_2. \end{array}$$

In subcase *(i)* we get $b_1; (b_2; b_3) \Rightarrow^p b_1'; (b_2; b_3)$, and since $p \sqsubseteq^\partial p$ and $(b_1'; b_2); b_3 \sqsubseteq b_1'; (b_2; b_3)$ (follows from **S1**) we get the required result. In subcase *(ii)* we have $b_1; (b_2; b_3) \Rightarrow^p b_2; b_3$, and since $\flat = b_2; b_3$ the result follows.

**The case S2.** First we study $(b_1 + b_2); b_3 \sqsubseteq b_1; b_3 + b_2; b_3$. So assume

$$(b_1 + b_2); b_3 \Rightarrow^p \flat.$$

There are two interesting subcases:

$$\begin{array}{llll} (i) & b_1 + b_2 \Rightarrow^p b' & \text{and} & \flat = b'; b_3, \\ (ii) & b_1 + b_2 \Rightarrow^p \surd & \text{and} & \flat = b_3. \end{array}$$

In subcase *(i)* the only interesting case is when $b_i \Rightarrow^p b'$ for $i = 1$ or $i = 2$. Then $b_i; b_3 \Rightarrow^p b'; b_3$, and hence $b_1; b_3 + b_2; b_3 \Rightarrow^p b'; b_3$. The result follows using **P1**. In subcase *(ii)* it must be the case that $b_i \Rightarrow^p \surd$ for $i = 1$ or $i = 2$. Then $b_i; b_3 \Rightarrow^p b_3$, and hence $b_1; b_3 + b_2; b_3 \Rightarrow^p b_3$, and the result follows.

Next we study $b_1; b_3 + b_2; b_3 \sqsubseteq (b_1 + b_2); b_3$. So assume

$$b_1; b_3 + b_2; b_3 \Rightarrow^p \flat.$$

The only interesting case is when $b_i; b_3 \Rightarrow^p \flat$ for $i = 1$ or $i = 2$. Now there are two interesting subcases:

$$\begin{array}{llll} (i) & b_i \Rightarrow^p b_i' & \text{and} & \flat = b_i'; b_3, \\ (ii) & b_i \Rightarrow^p \surd & \text{and} & \flat = b_3. \end{array}$$

In subcase *(i)* we get $b_1 + b_2 \Rightarrow^p b_i'$ and hence $(b_1 + b_2); b_3 \Rightarrow^p b_i'; b_3$. The result follows using **P1**. In subcase *(ii)* we get $b_1 + b_2 \Rightarrow^p \surd$ and hence $(b_1 + b_2); b_3 \Rightarrow^p b_2$. Again the result follows from **P1**.

**The case E1.** First we study $b \sqsubseteq \epsilon; b$. So assume $b \Rightarrow^p \flat$. Clearly $\epsilon; b \Rightarrow^\epsilon b \Rightarrow^p \flat$ and the result follows using **P1**.



Next we study $\epsilon; b \sqsubseteq b$. So assume $\epsilon; b \Rightarrow^p \flat$. There are two interesting subcases:

$$\begin{array}{llll} (i) & \epsilon \Rightarrow^\epsilon \epsilon & \text{and} & \flat = \epsilon; b \quad (\text{and } p = \epsilon), \\ (ii) & \epsilon \Rightarrow^\epsilon \surd & \text{and} & \flat = b \quad (\text{and } p = \epsilon). \end{array}$$

Clearly $b \Rightarrow^\epsilon b$. In subcase *(i)* the result follows from using the assumption **(E1)** that $\epsilon; b \sqsubseteq b$ and in subcase *(ii)* it follows using **P1**.

**The case E2.** First we study $b; \epsilon \sqsubseteq b$. So assume $b; \epsilon \Rightarrow^p \flat$. There are two interesting subcases:

$$\begin{array}{llll} (i) & b \Rightarrow^p b' & \text{and} & \flat = b'; \epsilon, \\ (ii) & b \Rightarrow^p \surd & \text{and} & \flat = \epsilon. \end{array}$$

In subcase *(i)* the result follows using $b'; \epsilon \sqsubseteq b'$ **(E2)**. In subcase *(ii)* it must be the case that $b = \epsilon$ and $p = \epsilon$. Clearly $\epsilon \Rightarrow^\epsilon \epsilon$ and the result follows because $\epsilon \sqsubseteq \epsilon$.

Next we study $b \sqsubseteq b; \epsilon$. So assume $b \Rightarrow^p \flat$. If $\flat \neq \surd$ then $b; \epsilon \Rightarrow^p \flat; \epsilon$, and the result follows because $\flat \sqsubseteq \flat; \epsilon$ **(E2)**. If $\flat = \surd$ then $p = \epsilon$ must be the case. So we have $b; \epsilon \Rightarrow^\epsilon \epsilon \Rightarrow^\epsilon \surd$, and since $\surd \sqsubseteq \surd$ the result follows.

**The case J1.** We shall study $b_i \sqsubseteq b_1 + b_2$ for $i = 1$ and $i = 2$. So assume $b_i \Rightarrow^p \flat$. But then $b_1 + b_2 \Rightarrow^p \flat$ and the result follows using **P1**.

**The case J2.** We shall study $b + b \sqsubseteq b$ since $b \sqsubseteq b + b$ follows from **J1**. So assume $b + b \Rightarrow^p \flat$. The only interesting case is when $b \Rightarrow^p \flat$ and the result follows immediately.

**The case R1.** First we study $\text{REC}\beta.b \sqsubseteq b[\beta \mapsto \text{REC}\beta.b]$. So assume

$$\text{REC}\beta.b \Rightarrow^p \flat.$$

The only interesting case is when $p = \epsilon$ and $\flat = b[\beta \mapsto \text{REC}\beta.b]$. Since $b[\beta \mapsto \text{REC}\beta.b] \Rightarrow^\epsilon b[\beta \mapsto \text{REC}\beta.b]$ the result follows using **P1**.

Next we study $b[\beta \mapsto \text{REC}\beta.b] \sqsubseteq \text{REC}\beta.b$. So assume

$$b[\beta \mapsto \text{REC}\beta.b] \Rightarrow^p \flat.$$

We have $\text{REC}\beta.b \Rightarrow^\epsilon b[\beta \mapsto \text{REC}\beta.b] \Rightarrow^p \flat$ and the result follows using **P1**.

**The case R2.** We first consider $\text{REC}\beta.b \sqsubseteq \text{REC}\beta'.b[\beta \mapsto \beta']$ for $\beta' \notin FV(b)$. So assume

$$\text{REC}\beta.b \Rightarrow^p \flat.$$

The only interesting case is when $p = \epsilon$ and $\flat = b[\beta \mapsto \text{REC}\beta.b]$. Then

$$\text{REC}\beta'.b[\beta \mapsto \beta'] \Rightarrow^p b[\beta \mapsto \text{REC}\beta'.b[\beta \mapsto \beta']].$$

That $b[\beta \mapsto \text{REC}\beta.b] \sqsubseteq b[\beta \mapsto \text{REC}\beta'.b[\beta \mapsto \beta']]$ then follows from **R1**, and $p \sqsubseteq^\partial p$ is immediate. The other direction is similar. □

# Appendix 7.C Semantic Properties of the Typing System

**Proof of Lemma 7.4.2.** We proceed by structural induction on $E$. The interesting case is when $E = \texttt{let } x = E_1 \texttt{ in } e_2$. Then we have

$$tenv \vdash E[e_0] : t \;\&\; b \quad \text{(because } tenv \vdash e_0 : t_0 \;\&\; b_0\text{)}$$

because

$$tenv \vdash E_1[e_0] : t_1 \;\&\; b_1 \quad \text{(because } tenv \vdash e_0 : t_0 \;\&\; b_0\text{)},$$
$$tenv[x \mapsto ts_1] \vdash e_2 : t \;\&\; b_2,$$
$$b_1; b_2 \sqsubseteq b,$$
$$ts_1 = \text{gen}(tenv, b_1)t_1.$$

By the induction hypothesis we get

$$tenv\, tenv' \vdash E_1[e_0] : t_1 \;\&\; b_1 \quad \text{(because } tenv\, tenv' \vdash e_0 : t_0 \;\&\; b_0\text{)},$$

and by Corollary 7.2.13 we get

$$tenv\, tenv' [x \mapsto ts_1] \vdash e_2 : t \;\&\; b_2,$$

and clearly

$$b_1; b_2 \sqsubseteq b,$$

and finally

$$ts_1 = \text{gen}(tenv\, tenv', b_1)t_1$$

because $FV(tenv') \subseteq FV(b_0)$ by assumption and $FV(b_0) \subseteq FV(b_1)$ by Fact 7.4.1. This completes the proof. □

**Proof of Lemma 7.4.4.** We proceed by structural induction on $E$. The more interesting cases are

**The case of** $E\ e$. Assume $tenv \vdash E[e_0]\ e : t \;\&\; b$. Then $b_1; b_2; b_0 \sqsubseteq b$ and the premises are $tenv \vdash E[e_0] : t' \to^{b_0} t \;\&\; b_1$ and $tenv \vdash e : t' \;\&\; b_2$. The induction hypothesis gives $tenv \vdash E[e_0'] : t' \to^{b_0} t \;\&\; b_1'$ where $p; b_1' \sqsubseteq b_1$. The rule for application gives $tenv \vdash E[e_0']\ e : t \;\&\; b_1'; b_2; b_0$. Using the laws **P2**, **C1**, and **S1** we get $p; b_1'; b_2; b_0 \sqsubseteq b$ as required.

**The case of** $w\ E$. Assume $tenv \vdash w\ E[e_0] : t \;\&\; b$. Then $b_1; b_2; b_0 \sqsubseteq b$ and the premises are $tenv \vdash w : t' \to^{b_0} t \;\&\; b_1$ and $tenv \vdash E[e_0] : t' \;\&\; b_2$. The induction hypothesis gives $tenv \vdash E[e_0'] : t' \;\&\; b_2'$ where $p; b_2' \sqsubseteq b_2$. From Fact 7.4.5 we have $\epsilon \sqsubseteq b_1$ and $tenv \vdash w : t' \to^{b_0} t \;\&\; \epsilon$ so the rule for application gives $tenv \vdash w\ E[e_0'] : t \;\&\; \epsilon; b_2'; b_0$. Using the laws **P1**, **P2**, **C1**, **S1**, **E1**, and **E2** we get $p; \epsilon; b_2'; b_0 \sqsubseteq b$ as required.

**The case of** $\texttt{let } x = E \texttt{ in } e$. Assume $tenv \vdash \texttt{let } x = E[e_0] \texttt{ in } e : t \;\&\; b$. Then $b_1; b_2 \sqsubseteq b$ and the premises are $tenv \vdash E[e_0] : t_1 \;\&\; b_1$ and $tenv[x \mapsto ts] \vdash e : t \;\&\; b_2$ where $ts = \text{gen}(tenv,\ b_1)\ t_1$. The induction hypothesis gives $tenv \vdash E[e_0'] : t_1 \;\&\; b_1'$ where $p; b_1' \sqsubseteq b_1$. Clearly $FV(b_1') \subseteq FV(b_1)$ follows from Fact 7.2.5; so $FV(t_1)\backslash(FV(tenv) \cup FV(b_1)) \subseteq FV(t_1)\backslash(FV(tenv) \cup FV(b_1'))$, and thereby $ts' \succeq ts$, where $ts' = \text{gen}(tenv, b_1')\ t_1$. Using Fact 7.2.8 we get $tenv[x \mapsto ts'] \vdash e : t \;\&\; b_2$ and then the rule for let-polymorphism gives $tenv \vdash \texttt{let } x = E[e_0'] \texttt{ in } e : t \;\&\; b_1'; b_2$. Using the laws **P1**, **P2**, **C1**, and **S1**, it follows that $p; b_1'; b_2 \sqsubseteq b$. □



**Proof of Proposition 7.4.6.** We proceed by induction on the inference for $e \to e'$. Most cases are similar, and so we only consider the case of let-polymorphism. For this assume that $E[\texttt{let } x = w \texttt{ in } e] \to E[e[x \mapsto w]]$ and

$$cenv \vdash E[\texttt{let } x = w \texttt{ in } e] : t \;\&\; b. \tag{1}$$

In this inference we can identify the node corresponding to the hole of $E$:

$$cenv \vdash \texttt{let } x = w \texttt{ in } e : t_2 \;\&\; b_1. \tag{2}$$

Its premises are $cenv \vdash w : t_1 \;\&\; b_2$ and

$$cenv[x \mapsto ts] \vdash e : t_2 \;\&\; b_3, \tag{3}$$

where $ts = \text{gen}(cenv, b_2)t_1$ and $b_2; b_3 \sqsubseteq b_1$. From Fact 7.4.5 we have $\epsilon \sqsubseteq b_2$ and

$$cenv \vdash w : t_1 \;\&\; \epsilon. \tag{4}$$

We shall now apply Lemma 7.2.14 to (3) and (4) and get $cenv \vdash e[x \mapsto w] : t_2 \;\&\; b_3$. The laws **P1**, **P2**, **C1**, and **E1** give $b_3 \sqsubseteq b_1$ so using Fact 7.2.9 we get

$$cenv \vdash e[x \mapsto w] : t_2 \;\&\; b_1. \tag{5}$$

Using Fact 7.4.3 with (1), (2), and (5) we get the required

$$cenv \vdash E[e[x \mapsto w]] : t \;\&\; b.$$

This completes the proof. □

**Proof of Proposition 7.4.7.** We proceed by induction on the inference for matching.

**The cases of send/receive.** Assume

$$(\langle \texttt{send } \langle \texttt{pair } ci\ w \rangle \rangle, \langle \texttt{receive } ci \rangle) \overset{(ci!, ci?)}{\rightsquigarrow} (w, w)$$

and

$$cenv \vdash \langle \texttt{send } \langle \texttt{pair } ci\ w \rangle \rangle : t_1 \texttt{ com } b_1 \;\&\; \epsilon,$$
$$cenv \vdash \langle \texttt{receive } ci \rangle : t_2 \texttt{ com } b_2 \;\&\; \epsilon.$$

Furthermore, assume $cenv\ ci = t$ chan $r$. Then the typing rule for weakly evaluated expressions gives $t_1 = t_2 = t$ and $cenv \vdash w : t\ \&\ b_0$, and furthermore $b_1 \sqsupseteq r!t$ and $b_2 \sqsupseteq r?t$. Now, $\mathcal{B}\ cenv\ ci! = r!t$ and $\mathcal{B}\ cenv\ ci? = r?t$, and so we only have to show $r!t;\epsilon \sqsubseteq b_1$ and $r?t;\epsilon \sqsubseteq b_2$, but this is immediate.

**The case of heads.** Assume $(\langle \texttt{choose}\ \langle \texttt{cons}\ w_1\ w_2\rangle\rangle, w_3) \overset{(d_1,d_3)}{\rightsquigarrow} (e_1, e_3)$ because $(w_1, w_3) \overset{(d_1,d_3)}{\rightsquigarrow} (e_1, e_3)$ and

$$cenv \vdash \langle \texttt{choose}\ \langle \texttt{cons}\ w_1\ w_2\rangle\rangle : t_1\ \texttt{com}\ b_1\ \&\ \epsilon,$$

$$cenv \vdash w_3 : t_3\ \texttt{com}\ b_3\ \&\ \epsilon.$$

From the typing rule for weakly evaluated expressions we get $cenv \vdash w_1 : t_1\ \texttt{com}\ b_0\ \&\ b$ for some $b \sqsupseteq \epsilon$ and where $b_1 \sqsupseteq b_0$. Using Fact 7.4.5 we have $cenv \vdash w_1 : t_1\ \texttt{com}\ b_0\ \&\ \epsilon$, and the induction hypothesis gives

$$cenv \vdash e_1 : t_1\ \&\ b_0' \quad \text{and} \quad cenv \vdash e_3 : t_3\ \&\ b_3',$$

where $(\mathcal{B}\ cenv\ d_1); b_0' \sqsubseteq b_0$ and $(\mathcal{B}\ cenv\ d_3); b_3' \sqsubseteq b_3$. Using the laws **P2** and **J1** we get $(\mathcal{B}\ cenv\ d_1); b_0' \sqsubseteq b_1$ and the result follows.

**The case of tails.** This is along the lines of the previous case and we omit the details.

**The case of** `wrap`. Assume $(\langle \texttt{wrap}\ \langle \texttt{pair}\ w_1\ w_2\rangle\rangle, w_3) \overset{(d_1,d_3)}{\rightsquigarrow} (w_2\ e_1, e_3)$ because $(w_1, w_3) \overset{(d_1,d_3)}{\rightsquigarrow} (e_1, e_3)$ and assume

$$cenv \vdash \langle \texttt{wrap}\ \langle \texttt{pair}\ w_1\ w_2\rangle\rangle : t_2\ \texttt{com}\ b_2\ \&\ \epsilon,$$

$$cenv \vdash w_3 : t_3\ \texttt{com}\ b_3\ \&\ \epsilon.$$

From the typing rule for weakly evaluated expressions we get

$$cenv \vdash w_1 : t_1\ \texttt{com}\ b_1\ \&\ b \quad \text{and} \quad cenv \vdash w_2 : t_1 \rightarrow^{b_0} t_2\ \&\ b'$$

and $b_2 \sqsupseteq b_1; b_0$. Using Fact 7.4.5 we get

$$cenv \vdash w_1 : t_1\ \texttt{com}\ b_1\ \&\ \epsilon \quad \text{and} \quad cenv \vdash w_2 : t_1 \rightarrow^{b_0} t_2\ \&\ \epsilon\ .$$

Then the induction hypothesis gives

$$cenv \vdash e_1 : t_1\ \&\ b_1' \quad \text{and} \quad cenv \vdash e_3 : t_3\ \&\ b_3',$$

where $(\mathcal{B}\ cenv\ d_1); b_1' \sqsubseteq b_1$ and $(\mathcal{B}\ cenv\ d_3); b_3' \sqsubseteq b_3$. The rule for function application then gives

$$cenv \vdash w_2\ e_1 : t_2\ \&\ \epsilon; b_1'; b_0,$$

and we have to show $(\mathcal{B}\ cenv\ d_1); \epsilon; b_1'; b_0 \sqsubseteq b_2$. But this is immediate.

**The case of swop.** This case is straightforward and we omit the details. This completes the proof. □



**Proof of Theorem 7.4.8.** We proceed by induction on the inference for concurrent evaluation.

**The case of sequential evaluation.** Assume that

$$CI\ \&\ PP[pi \mapsto E[e]] \longrightarrow^{\epsilon}_{pi} CI\ \&\ PP[pi \mapsto E[e']]$$

because $E[e] \rightarrow E[e']$. Furthermore, assume that $CI\&PP[pi \mapsto E[e]]$ is $cenv$-described by $PB[pi \mapsto b]$. In particular this means that

$$cenv \vdash E[e] : t\ \&\ b$$

for some $t$. From Proposition 7.4.6 we get

$$cenv \vdash E[e'] : t\ \&\ b.$$

Clearly $b \Rightarrow^{\epsilon} b$, and thereby

$$PB[pi \mapsto b] \Longrightarrow^{\widehat{\epsilon}}_{pi} PB[pi \mapsto b].$$

We have $\mathcal{A}\ (cenv, PB[pi \mapsto b])\ \epsilon = \epsilon$. Taking $PB' = PB[pi \mapsto b]$ we get that $CI\&PP[pi \mapsto E[e']]$ is $cenv$-described by $PB'$ as required.

**The case of channel allocation.** Assume that

$$CI\ \&\ PP[pi \mapsto E[\texttt{channel}_l\ ()]] \longrightarrow^{\texttt{CHAN}_l\ ci}_{pi}$$
$$CI\ \cup\ \{ci\}\ \&\ PP[pi \mapsto E[ci]],$$

where $ci \notin CI$. Furthermore, assume that $CI\&PP[pi \mapsto E[\texttt{channel}_l\ ()]]$ is $cenv$-described by $PB[pi \mapsto b]$. In particular this means that

$$cenv \vdash E[\texttt{channel}_l\ ()] : t\ \&\ b \tag{1}$$

for some $t$. In this inference we can identify the node corresponding to the hole of $E$ and it will be of the form

$$cenv \vdash \texttt{channel}_l\ () : t'\ \texttt{chan}\ r\ \&\ b_0, \tag{2}$$

where $l \in r$ and $b_1; b_2; t'$ CHAN $r \sqsubseteq b_0$, where $\epsilon \sqsubseteq b_1$ and $\epsilon \sqsubseteq b_2$ (follows from Fact 7.4.5). Now define $cenv' = cenv[ci \mapsto t'\ \texttt{chan}\ r]$. Then we can replace $cenv$ in (2) and (1) by $cenv'$ and thus obtain (2′) and (1′); by Lemma 7.4.2 we still have (1′) because (2′). From (2′) we then get

$$cenv' \vdash ci : t'\ \texttt{chan}\ r\ \&\ \epsilon.$$

Also, $t'$ CHAN $r$; $\epsilon \sqsubseteq b_0$ follows from the laws **P1**, **P2**, **C1**, **E1**, **E2**, and **J1**; so we can apply Lemma 7.4.4 and get

$$cenv' \vdash E[ci] : t\ \&\ b', \tag{3}$$

where $t'$ CHAN $r$; $b' \sqsubseteq b$. Clearly $t'$ CHAN $r; b' \Rightarrow^{t'\ \texttt{CHAN}\ r} b'$, and using that $\sqsubseteq$ is a simulation (Proposition 7.3.2) we get $b \Rightarrow^{\widehat{p}} \tilde{b}$ for some $p$ and $\tilde{b}$ satisfying $t'$ CHAN $r \sqsubseteq^{\partial} p$ and $b' \sqsubseteq \tilde{b}$. Now $p = t'$ CHAN $r$ must be the case; so

$$PB[pi \mapsto b] \Longrightarrow_{pi}^{t' \widehat{\text{CHAN } r}} PB[pi \mapsto \tilde{b}].$$

Clearly $\mathcal{A}\ (cenv', PB[pi \mapsto \tilde{b}])\ \text{CHAN}\ ci = t'\ \text{CHAN}\ r$ as required. Taking $PB' = PB[pi \mapsto \tilde{b}]$ we get from (3) and Fact 7.2.9 (and $b' \sqsubseteq \tilde{b}$) that $CI \cup \{ci\}\&PP[pi \mapsto E[ci]]$ is $cenv'$-described by $PB'$.

**The case of process creation.** Assume that

$$CI\ \&\ PP[pi_1 \mapsto E[\texttt{fork}_\pi\ w]] \longrightarrow_{pi_1,pi_2}^{\text{FORK}_\pi\ pi_2},$$
$$CI\ \&\ PP[pi_1 \mapsto E[()]][pi_2 \mapsto w()],$$

where $pi_2 \notin \text{Dom}(PP) \cup \{pi_1\}$. Furthermore, assume that $CI\&PP[pi_1 \mapsto E[\texttt{fork}_\pi\ w]]$ is $cenv$-described by $PB[pi_1 \mapsto b_1]$. In particular this means that

$$cenv \vdash E[\texttt{fork}_\pi\ w] : t_1\ \&\ b_1$$

for some $t_1$. In this inference we can identify the node corresponding to the hole of $E$. It has the form

$$cenv \vdash\ \texttt{fork}_\pi\ w : \texttt{unit}\ \&\ b_0, \tag{1}$$

where $b_1; b_2; \text{FORK}_\pi\ b \sqsubseteq b_0$ for some $b$ and where $\epsilon \sqsubseteq b_1$ and $\epsilon \sqsubseteq b_2$ follows from Fact 7.4.5. We also have

$$cenv \vdash ()\ :\ \texttt{unit}\ \&\ \epsilon,$$

and $(\text{FORK}_\pi\ b); \epsilon \sqsubseteq b_0$ follows from the laws **P1**, **P2**, **C1**, **E1**, **E2**, and **J1**. We can then apply Lemma 7.4.4 and get

$$cenv \vdash E[()] : t_1\ \&\ b_1', \tag{2}$$

where $(\text{FORK}_\pi\ b); b_1' \sqsubseteq b_1$. Clearly $(\text{FORK}_\pi\ b); b_1' \Rightarrow^{\text{FORK}_\pi\ b} b_1'$, and using that $\sqsubseteq$ is a simulation (Proposition 7.3.2) we get $b_1 \Rightarrow^{\widehat{p}} \tilde{b_1}$ for some $p$ and $\tilde{b_1}$ with $\text{FORK}_\pi\ b \sqsubseteq^\partial p$ and $b_1' \sqsubseteq \tilde{b_1}$. Now it follows that $p = \text{FORK}_\pi\ b'$ for some $b'$ where $b \sqsubseteq b'$. Also we have

$$PB[pi_1 \mapsto b_1] \Longrightarrow_{pi_1,pi_2}^{\widehat{\text{FORK}_\pi\ b'}} PB[pi_1 \mapsto \tilde{b_1}][pi_2 \mapsto b'].$$

We have $\mathcal{A}\ (cenv, PB[pi_1 \mapsto \tilde{b_1}][pi_2 \mapsto b'])\ \text{FORK}_\pi\ pi_2 = \text{FORK}_\pi\ b'$ as required. We shall now take $PB' = PB[pi_1 \mapsto \tilde{b_1}][pi_2 \mapsto b']$. From (1) and Fact 7.4.5 we get

$$cenv \vdash w : \texttt{unit}\ \rightarrow^b t\ \&\ \epsilon$$

for some type $t$, and thereby

$$cenv \vdash w\ ()\colon t\ \&\ b.$$



Using this and (2) together with Fact 7.2.9 (and $b_1' \sqsubseteq \tilde{b_1}$ and $b \sqsubseteq b'$) we get that $CI\&PP[pi_1 \mapsto E[()][pi_2 \mapsto w()]$ is $cenv$-described by $PB'$.

**The case of matching.** Assume that

$$CI\ \&\ PP[pi_1 \mapsto E_1[\text{sync}\ w_1]][pi_2 \mapsto E_2[\text{sync}\ w_2]] \longrightarrow_{pi_1,pi_2}^{(ci!,ci?)},$$
$$CI\ \&\ PP[pi_1 \mapsto E_1[e_1]][pi_2 \mapsto E_2[e_2]],$$

because $(w_1, w_2) \overset{(ci!,ci?)}{\rightsquigarrow} (e_1, e_2)$. Furthermore, assume that $CI\&PP[pi_1 \mapsto E_1[\text{sync}\ w_1]]\ [pi_2 \mapsto E_2[\text{sync}\ w_2]]$ is $cenv$-described by $PB[pi_1 \mapsto b_1][pi_2 \mapsto b_2]$. In particular this means that

$$cenv \vdash E_1[\text{sync}\ w_1] : t_1\ \&\ b_1 \text{ and } cenv \vdash E_2[\text{sync}\ w_2] : t_2\ \&\ b_2$$

for some $t_1$ and $t_2$. In these inferences we can identify the node corresponding to the hole of $E_1$ and $E_2$, respectively, and they have the forms

$$cenv \vdash \text{sync}\ w_1 : t_1'\ \&\ \overline{b_1} \text{ and } cenv \vdash \text{sync}\ w_2 : t_2'\ \&\ \overline{b_2},$$

where $b_{11}; b_{12}; b_{01} \sqsubseteq \overline{b_1}$ and $b_{21}; b_{22}; b_{02} \sqsubseteq \overline{b_2}$ for some $b_{01}$ and $b_{02}$. The premises include

$$cenv \vdash w_1 : t_1'\ \texttt{com}\ b_{01}\ \&\ b_{12} \text{ and } cenv \vdash w_2 : t_2'\ \texttt{com}\ b_{02}\ \&\ b_{22},$$

and from Fact 7.4.5 we get $\epsilon \sqsubseteq b_{11}$, $\epsilon \sqsubseteq b_{12}$, $\epsilon \sqsubseteq b_{21}$, $\epsilon \sqsubseteq b_{22}$, and

$$cenv \vdash w_1 : t_1'\ \texttt{com}\ b_{01}\ \&\ \epsilon \text{ and } cenv \vdash w_2 : t_2'\ \texttt{com}\ b_{02}\ \&\ \epsilon.$$

Using Proposition 7.4.7 we get

$$cenv \vdash e_1 : t_1'\ \&\ b_{01}' \text{ and } cenv \vdash e_2 : t_2'\ \&\ b_{02}',$$

where $(\mathcal{B}\ cenv\ ci!); b_{01}' \sqsubseteq b_{01}$ and $(\mathcal{B}\ cenv\ ci?); b_{02}' \sqsubseteq b_{02}$. Using the laws **P1**, **P2**, **C1**, **E1**, and **J1** we get $(\mathcal{B}\ cenv\ ci!); b_{01}' \sqsubseteq \overline{b_1}$ and $(\mathcal{B}\ cenv\ ci?); b_{02}' \sqsubseteq \overline{b_2}$. We can now apply Lemma 7.4.4 and get

$$cenv \vdash E_1[e_1] : t_1\ \&\ b_1' \text{ and } cenv \vdash E_2[e_2] : t_2\ \&\ b_2', \tag{1}$$

where $(\mathcal{B}\ cenv\ ci!); b_1' \sqsubseteq b_1$ and $(\mathcal{B}\ cenv\ ci?); b_2' \sqsubseteq b_2$. Clearly $(\mathcal{B}\ cenv\ ci!); b_1' \Rightarrow^{(\mathcal{B}\ cenv\ ci!)} b_1'$ and $(\mathcal{B}\ cenv\ ci?); b_2' \Rightarrow^{(\mathcal{B}\ cenv\ ci?)} b_2'$. Using that $\sqsubseteq$ is a simulation (Proposition 7.3.2), we get $b_1 \Rightarrow^{\widehat{p_1}} \tilde{b_1}$ and $b_2 \Rightarrow^{\widehat{p_2}} \tilde{b_2}$, where $(\mathcal{B}\ cenv\ ci!) \sqsubseteq^\partial p_1$, $(\mathcal{B}\ cenv\ ci?) \sqsubseteq^\partial p_2$, $b_1' \sqsubseteq \tilde{b_1}$, and $b_2' \sqsubseteq \tilde{b_2}$. It must be the case that $(p_1, p_2) = ((\mathcal{B}\ cenv\ ci!), (\mathcal{B}\ cenv\ ci?)) = (r!t, r?t)$ for some $r$ and $t$; so we get

$$PB[pi_1 \mapsto b_1][pi_2 \mapsto b_2] \Longrightarrow_{pi_1,pi_2}^{\widehat{(r!t,r?t)}} PB[pi_1 \mapsto \tilde{b_1}][pi_2 \mapsto \tilde{b_2}].$$

Now $\mathcal{A}\ (cenv, PB[pi_1 \mapsto \tilde{b_1}][pi_2 \mapsto \tilde{b_2}])\ (ci!, ci?) = (r!t, r?t)$ as required. Taking $PB' = PB[pi_1 \mapsto \tilde{b_1}][pi_2 \mapsto \tilde{b_2}]$ we get that

$$CI\&PP[pi_1 \mapsto E_1[e_1]][pi_2 \mapsto E_2[e_2]] \text{ is } cenv\text{-described by } PB',$$

using (1) together with Fact 7.2.9 (and $b_1' \sqsubseteq \tilde{b_1}$ and $b_2' \sqsubseteq \tilde{b_2}$). □

# Appendix 7.D Decidability Issues Concerning the Orderings

**Proof of Lemma 7.5.2.** We prove the two implications separately. Proving "$\Leftarrow$": Let $b' \sqsubseteq\!\!\!\sim\, b''$ and let $w \in \mathcal{L}(b')$. Then there exists

$$b' \Rightarrow^{p'_1} \flat'_1 \ldots \Rightarrow^{p'_n} \flat'_n \text{ with } \flat'_n = \surd$$

such that $w = p'_1 \ldots p'_n$ (and where some $p'_i$ may be $\epsilon$ and hence disappear). By $b' \sqsubseteq\!\!\!\sim\, b''$ we get

$$b'' \Rightarrow^{\widehat{p''_1}} \flat''_1 \ldots \Rightarrow^{\widehat{p''_n}} \flat''_n,$$

with $\flat'_i \sqsubseteq\!\!\!\sim\, \flat''_i$ and $p'_i \sqsubseteq^{\partial}\!\!\!\sim\, p''_i$ for all $i$. As each $p'_i$ is either $\epsilon$ or of the form $\mathtt{1} \mathbin{!} t$ it follows that $p''_i = p'_i$ for all $i$. Also, $\flat''_n$ equals $\surd$. It follows that $w \in \mathcal{L}(b'')$.

Proving "$\Rightarrow$": We define

$$S = \{(b_1, b_2) \mid \mathcal{L}(b_1) \subseteq \mathcal{L}(b_2), \vdash_D b_1, \vdash_D b_2, \models_N b_1, \models_N b_2\} \cup \{(\surd, \surd)\}$$

and show that $S$ is a simulation on closed behaviors; since $b' \; S \; b''$ the desired $b' \sqsubseteq\!\!\!\sim\, b''$ follows. First we note that $\surd \; S \; \flat$ if and only if $\flat = \surd$. Next let $(b_1, b_2) \in S \backslash \{(\surd, \surd)\}$ and suppose $b_1 \Rightarrow^{p_1} \flat_{11}$; if $\flat_{11} \neq \surd$ we have (using Lemma 7.5.1) that $\vdash_D p_1, \vdash_D \flat_{11}, \models_N p_1, \models_N \flat_{11}$. We now perform a case analysis.

*(i)* If $p_1 = \epsilon$ and $\flat_{11} \neq \surd$ we have $b_2 \Longrightarrow^{\widehat{\epsilon}} b_2$ with $(\flat_{11}, b_2) \in S$ and $(\epsilon, \epsilon) \in S^{\partial}$, where we have used Lemma 7.5.1.

*(ii)* If $p_1 = \epsilon$ and $\flat_{11} = \surd$ we have $\epsilon \in \mathcal{L}(b_1)$, and hence $\epsilon \in \mathcal{L}(b_2)$, so that $b_2 \Rightarrow^{\widehat{\epsilon}} \surd$ with $(\surd, \surd) \in S$ and $(\epsilon, \epsilon) \in S^{\partial}$.

*(iii)* If $p_1 = \mathtt{1} \mathbin{!} t$ (for some $\mathtt{1}$ and $t$) we have $\flat_{11} \stackrel{w}{\Longrightarrow}{}^{*} \surd$, so that $p_1 w \in \mathcal{L}(b_1)$ and $p_1 w \in \mathcal{L}(b_2)$. Hence $b_2 \Longrightarrow^{\widehat{p_1}} b_{21} \stackrel{w}{\Longrightarrow}{}^{*} \surd$. From Lemma 7.5.1 we have

$$\mathcal{L}(\flat_{11}) = \{w \mid p_1 w \in \mathcal{L}(b_1)\},$$
$$\mathcal{L}(b_{21}) = \{w \mid p_1 w \in \mathcal{L}(b_2)\},$$

and since $\mathcal{L}(b_1) \subseteq \mathcal{L}(b_2)$ we get $\mathcal{L}(\flat_{11}) \subseteq \mathcal{L}(b_{21})$. Using Lemma 7.5.1 we have $(\flat_{11}, b_{21}) \in S$ and $(p_1, p_1) \in S^{\partial}$. □

**Proof of Lemma 7.5.8.** We inspect each of the laws:

**The case P1:** We shall prove the (stronger) result

$$bn \text{ ord } bn, \tag{1}$$

$$bs \text{ ord}_s \; bs, \tag{2}$$

$$bp \text{ ord}_p \; bp. \tag{3}$$



We proceed by induction on the *size* of the behaviors $bn, bs, bp$ proving (3), (2), and (1) in order.

First we prove (3). If $bp$ is one of $r!t$, $r?t$, $t$ CHAN $r$, and $\beta$, then the result is immediate from the definition of $\text{ord}_p$. If $bp = \text{FORK}_\pi \; bn$, then the induction hypothesis gives $bn$ ord $bn$ and the result follows from the definition of $\text{ord}_p$. The case where $bp = \text{REC}\beta.bn$ is similar.

Then we prove (2). If $bs = \epsilon$ the result is immediate from the definition of $\text{ord}_s$. If $bs = bp$ then the result follows from (3) and the definition of $\text{ord}_s$. So assume $bs = bp; bn$. Then the induction hypothesis gives $bp$ ord $bp$ and $bn$ ord $bn$ and the result follows from the definition of $\text{ord}_s$.

Finally we prove (1). If $bn = bs$ the result follows from (2) and the definition of ord. So assume $bn = bn' + bn''$. Then the induction hypothesis gives $bn'$ ord $bn'$ and $bn''$ ord $bn''$. We then get $bn'$ ord $bn' + bn''$ and $bn''$ ord $bn' + bn''$. From the definition of ord it now follows that $bn' + bn''$ ord $bn' + bn''$.

**The case P2.** We shall prove the (stronger) result

$$bn_1 \text{ ord } bn_2 \text{ and } bn_2 \text{ ord } bn_3 \text{ imply } bn_1 \text{ ord } bn_3 \tag{1}$$

$$bs_1 \text{ ord}_s \; bs_2 \text{ and } bs_2 \text{ ord}_s \; bs_3 \text{ imply } bs_1 \text{ ord}_s \; bs_3 \tag{2}$$

$$bp_1 \text{ ord}_p \; bp_2 \text{ and } bp_2 \text{ ord}_p \; bp_3 \text{ imply } bp_1 \text{ ord}_p \; bp_3 \tag{3}$$

We proceed by induction on the *size* of the behaviors $bn_1, bs_1, bp_1$, proving (3), (2), and (1) in order.

First we prove (3). If $bp_1$ is one of $r!t$, $r?t$, $t$ CHAN $r$, and $\beta$, then $bp_2 = bp_1$ must be the case because of the definition of $\text{ord}_p$, and similarly $bp_3 = bp_2$ also must be the case. The result now follows from the definition of $\text{ord}_p$. If $bp_1 = \text{FORK}_\pi \; bn_1$ then $bp_2 = \text{FORK}_\pi \; bn_2$ must be the case because of the definition of $\text{ord}_p$, and furthermore $bn_1$ ord $bn_2$ holds. Similarly, $bp_3 = \text{FORK}_\pi \; bn_3$ must be the case and $bn_2$ ord $bn_3$ holds. The induction hypothesis gives $bn_1$ ord $bn_3$, and the result follows using the definition of $\text{ord}_p$. The case where $bp_1 = \text{REC}\beta.bn_1$ is similar.

Then we prove (2). If $bs_1 = \epsilon$ then the definition of $\text{ord}_s$ gives that $bs_2 = \epsilon$ must be the case, and similarly $bs_3 = \epsilon$ must be the case. Then the result follows trivially from the definition of $\text{ord}_s$. If $bs_1 = bp_1$ then $bs_2 = bp_2$ must be the case because of the definition of $\text{ord}_s$, and similarly $bs_3 = bp_3$ must be the case. Then the result follows from (3) and the definition of $\text{ord}_s$. So assume $bs_1 = bp_1; bn_1$. Then $bs_2 = bp_2; bn_2$ must be the case because of the definition of $\text{ord}_s$, and furthermore $bp_1$ ord $bp_2$ and $bn_1$ ord $bn_2$. Similarly, $bs_3 = bp_3; bn_3$ must be the case, and $bp_2$ ord $bp_3$ and $bn_2$ ord $bn_3$. The induction hypothesis gives $bp_1$ ord $bp_3$ and $bn_1$ ord $bn_3$ and the result follows using the definition of $\text{ord}_s$.

Finally we prove (1). If $bn_1 = bs_1$ then Lemma 7.5.7 gives that there exists $bs_2 \in \Sigma(bn_2)$ such that $bs_1$ $\text{ord}_s$ $bs_2$. But then there exists $bs_3 \in \Sigma(bn_3)$ such that $bs_2$ $\text{ord}_s$ $bs_3$. From (2) we get $bs_1$ $\text{ord}_s$ $bs_3$ and we have

$bs_1$ ord $bn_3$ using Lemma 7.5.7. Next assume $bn_1 = bn_1' + bn_1''$. From $bn_1' + bn_1''$ ord $bn_2$ we get $bn_1'$ ord $bn_2$ and $bn_1''$ ord $bn_2$ and the induction hypothesis gives $bn_1'$ ord $bn_3$ and $bn_1''$ ord $bn_3$. But then the result follows using the definition of ord.

**The case C1.** Using Lemma 7.5.7 it suffices to prove the (stronger) result

$$\text{if } bn_1 \text{ ord } bn_2 \text{ and } bn_3 \text{ ord } bn_4 \text{ then } \mathsf{seq}(bn_1, bn_3) \text{ ord } \mathsf{seq}(bn_2, bn_4) \quad (1)$$

$$\text{if } bs_1 \text{ ord}_s\ bs_2 \text{ and } bn_3 \text{ ord } bn_4 \text{ then } \mathsf{seq}(bs_1, bn_3) \text{ ord } \mathsf{seq}(bs_2, bn_4) \quad (2)$$

To do this we need

$$\Sigma(\mathsf{seq}(bn, bn_0)) = \bigcup\{\Sigma(\mathsf{seq}(bs, bn_0)) \mid bs \in \Sigma(bn)\}, \qquad (*)$$

which can be proved by a straightforward structural induction on $bn$. The proofs of (1) and (2) proceed by induction on the *size* of the behaviors $bn_1, bs_1$, proving (2) and (1) in order.

First we prove (2). To prove $\mathsf{seq}(bs_1, bn_3)$ ord $\mathsf{seq}(bs_2, bn_4)$ it is sufficient to prove that whenever $bs \in \Sigma(\mathsf{seq}(bs_1, bn_3))$ then there exists $bs' \in \Sigma(\mathsf{seq}(bs_2, bn_4))$ such that $bs$ ord$_s$ $bs'$, and the result follows from Lemma 7.5.7. So assume $bs \in \Sigma(\mathsf{seq}(bs_1, bn_3))$. If $bs_1 = \epsilon$ then $bs_2 = \epsilon$ must be the case because of the definition of ord$_s$, and furthermore $bs \in \Sigma(bn_3)$ must be the case. From $bn_3$ ord $bn_4$ and Lemma 7.5.7 we get that there exists $bs' \in \Sigma(bn_4)$ such that $bs$ ord$_s$ $bs'$. Since $bs' \in \Sigma(\mathsf{seq}(\epsilon, bn_4))$ the result follows. Next assume $bs_1 = bp_1$. Then $bs_2 = bp_2$ must be the case because of the definition of ord$_s$, and furthermore $bs = bp_1; bn_3$ must be the case. Using the definition of ord$_s$ we get $bp_1; bn_3$ ord$_s$ $bp_2; bn_4$ and the result follows since $bp_2; bn_4 \in \Sigma(\mathsf{seq}(bp_2, bn_4))$. Finally, assume $bs_1 = bp_1; bn_1$. Then $bs_2 = bp_2; bn_2$ must be the case because of the definition of ord$_s$, and furthermore $bp_1$ ord $bp_2$ and $bn_1$ ord $bn_2$. From $bs \in \Sigma(\mathsf{seq}(bp_1; bn_1, bn_3))$ we see that $bs = bp_1; \mathsf{seq}(bn_1, bn_3)$. The induction hypothesis gives $\mathsf{seq}(bn_1, bn_3)$ ord $\mathsf{seq}(bn_2, bn_4)$. Thus we have $bp_1; \mathsf{seq}(bn_1, bn_3)$ ord$_s$ $bp_2; \mathsf{seq}(bn_2, bn_4)$ using the definition of ord$_s$ and since $bp_2; \mathsf{seq}(bn_2, bn_4) \in \Sigma(\mathsf{seq}(bp_2; bn_2, bn_4))$, this is the required result.

Then we prove (1). To prove $\mathsf{seq}(bn_1, bn_3)$ ord $\mathsf{seq}(bn_2, bn_4)$ it suffices to prove that whenever $bs \in \Sigma(\mathsf{seq}(bn_1, bn_3))$ then there exists $bs' \in \Sigma(\mathsf{seq}(bn_2, bn_4))$ such that $bs$ ord$_s$ $bs'$, and the result follows from Lemma 7.5.7. So assume $bs \in \Sigma(\mathsf{seq}(bn_1, bn_3))$. Using (*) this means that $bs \in \Sigma(\mathsf{seq}(bs_1, bn_3))$ for some $bs_1 \in \Sigma(bn_1)$. From $bn_1$ ord $bn_2$ and Lemma 7.5.7 we get that $bs_1$ ord$_s$ $bs_2$ for some $bs_2 \in \Sigma(bn_2)$. Then (2) and Lemma 7.5.7 gives that there is $bs' \in \Sigma(\mathsf{seq}(bs_2, bn_4))$ such that $bs$ ord$_s$ $bs'$. But $bs' \in \Sigma(\mathsf{seq}(bn_2, bn_4))$ due to (*), and the result follows.

**The case C2.** We shall prove

$$\text{if } bn_1 \text{ ord } bn_2 \text{ and } bn_3 \text{ ord } bn_4 \text{ then } bn_1 + bn_3 \text{ ord } bn_2 + bn_4.$$

Using Lemma 7.5.7 it is sufficient to prove that whenever $bs \in \Sigma(bn_1 + bn_3)$ then there exists $bs' \in \Sigma(bn_2 + bn_4)$ such that $bs$ ord$_s$ $bs'$. It is immediate that

$$\Sigma(bn_i + bn_{i+2}) = \Sigma(bn_i) \cup \Sigma(bn_{i+2})$$

for $i = 1, 2$. So given $bs \in \Sigma(bn_1 + bn_3)$ we have $bs \in \Sigma(bn_i)$, and Lemma 7.5.7 together with $bn_i$ ord $bn_{i+1}$ gives that there exists $bs' \in \Sigma(bn_{i+1})$ such that $bs$ ord$_s$ $bs'$. Since $bs' \in \Sigma(bn_2 + bn_4)$ the result follows.

**The case C3.** We shall prove

$$\text{if } bn_1 \text{ ord } bn_2 \text{ then } \textsc{fork}_\pi\ bn_1 \text{ ord } \textsc{fork}_\pi\ bn_2.$$

But this follows from the definitions of ord, ord$_s$, and ord$_p$.

**The case C4.** We shall prove

$$\text{if } bn_1 \text{ ord } bn_2 \text{ then } \textsc{rec}\beta.\ bn_1 \text{ ord } \textsc{rec}\beta.\ bn_2.$$

But this follows from the definitions of ord, ord$_s$, and ord$_p$.

**The case S1.** We shall prove

$$\mathsf{seq}(bn_1, \mathsf{seq}(bn_2, bn_3)) \text{ ord } \mathsf{seq}(\mathsf{seq}(bn_1, bn_2), bn_3),$$

$$\mathsf{seq}(\mathsf{seq}(bn_1, bn_2), bn_3) \text{ ord } \mathsf{seq}(bn_1, \mathsf{seq}(bn_2, bn_3)).$$

Below we show that

$$\mathsf{seq}(bn_1, \mathsf{seq}(bn_2, bn_3)) = \mathsf{seq}(\mathsf{seq}(bn_1, bn_2), bn_3). \qquad (*)$$

Since ord fulfills the law **P1** we have $bn$ ord $bn$ for all $bn$, and the result follows.

Turning to the proof of (*), note that the result is immediate if $bn_3 = \epsilon$, so henceforth assume that $bn_3 \neq \epsilon$. In a similar way note that the result is immediate if $bn_2 = \epsilon$; so henceforth assume that also $bn_2 \neq \epsilon$. We now proceed by structural induction on $bn_1$.

First assume that $bn_1 = bs_1$. If $bs_1 = \epsilon$ then the result holds trivially since

$$\begin{aligned} \mathsf{seq}(\epsilon, \mathsf{seq}(bn_2, bn_3)) &= \mathsf{seq}(bn_2, bn_3) \\ &= \mathsf{seq}(\mathsf{seq}(\epsilon, bn_2), bn_3). \end{aligned}$$

If $bs_1 = bp_1$ then the result follows from

$$\begin{aligned} \mathsf{seq}(bp_1, \mathsf{seq}(bn_2, bn_3)) &= bp_1; \mathsf{seq}(bn_2, bn_3) \\ &= \mathsf{seq}(\mathsf{seq}(bp_1, bn_2), bn_3). \end{aligned}$$

Finally, if $bs_1 = bp_1; bn_1'$ then the result follows from

$$\begin{aligned} \mathsf{seq}(bp_1; bn_1', \mathsf{seq}(bn_2, bn_3)) &= bp_1; \mathsf{seq}(bn_1', \mathsf{seq}(bn_2, bn_3)) \\ &= bp_1; \mathsf{seq}(\mathsf{seq}(bn_1', bn_2), bn_3) \\ &= \mathsf{seq}(bp_1; \mathsf{seq}(bn_1', bn_2), bn_3) \\ &= \mathsf{seq}(\mathsf{seq}(bp_1; bn_1', bn_2), bn_3), \end{aligned}$$

where we have used the induction hypothesis to obtain the second equality.

Next assume $bn_1 = bn_1' + bn_1''$. Then

$$\begin{aligned}
&\mathsf{seq}(bn_1' + bn_1'', \mathsf{seq}(bn_2, bn_3)) \\
&= \mathsf{seq}(bn_1', \mathsf{seq}(bn_2, bn_3)) + \mathsf{seq}(bn_1'', \mathsf{seq}(bn_2, bn_3)) \\
&= \mathsf{seq}(\mathsf{seq}(bn_1', bn_2), bn_3) + \mathsf{seq}(\mathsf{seq}(bn_1'', bn_2), bn_3) \\
&= \mathsf{seq}(\mathsf{seq}(bn_1', bn_2) + \mathsf{seq}(bn_1'', bn_2), bn_3) \\
&= \mathsf{seq}(\mathsf{seq}(bn_1' + bn_1'', bn_2), bn_3),
\end{aligned}$$

where we have used the induction hypothesis to obtain the second equality.

**The case S2.** We shall prove

$$\mathsf{seq}(bn_1 + bn_2, bn_3) \text{ ord } \mathsf{seq}(bn_1, bn_3) + \mathsf{seq}(bn_2, bn_3),$$

$$\mathsf{seq}(bn_1, bn_3) + \mathsf{seq}(bn_2, bn_3) \text{ ord } \mathsf{seq}(bn_1 + bn_2, bn_3).$$

Clearly we have

$$\mathsf{seq}(bn_1 + bn_2, bn_3) = \mathsf{seq}(bn_1, bn_3) + \mathsf{seq}(bn_2, bn_3).$$

Since ord fulfills the law **P1** we have $bn$ ord $bn$ for all $bn$ and the result follows.

**The case E1.** We shall prove

$$\mathsf{seq}(\epsilon, bn) \text{ ord } bn \text{ and } bn \text{ ord } \mathsf{seq}(\epsilon, bn).$$

We have $\mathsf{seq}(\epsilon, bn) = bn$ and the result follows from the law **P1**.

**The case E2.** We shall prove

$$\mathsf{seq}(bn, \epsilon) \text{ ord } bn \text{ and } bn \text{ ord } \mathsf{seq}(bn, \epsilon).$$

We have $\mathsf{seq}(bn, \epsilon) = bn$ and the result follows from the law **P1**.

**The case J1.** We shall prove

$$bn_i \text{ ord } bn_1 + bn_2$$

for $i = 1, 2$. We have $\Sigma(bn_i) \subseteq \Sigma(bn_1 + bn_2)$ and the result follows from the law **P1** (for $\mathsf{ord}_s$) and Lemma 7.5.7.

**The case J2.** We shall prove

$$bn \text{ ord } bn + bn \text{ and } bn + bn \text{ ord } bn.$$

We have $\Sigma(bn) = \Sigma(bn + bn)$ and the result follows from the law **P1** (for $\mathsf{ord}_s$) and Lemma 7.5.7.

**The case R2.** We shall prove

$$\text{REC}\beta.bn \text{ ord } \text{REC}\beta'.bn[\beta \mapsto \beta'], \tag{1}$$

$$\text{REC}\beta'.bn[\beta \mapsto \beta'] \text{ ord } \text{REC}\beta.bn, \tag{2}$$



for $\beta' \notin FV(bn)$. Clearly $bn[\beta \mapsto \beta''] = (bn[\beta \mapsto \beta'])[\beta' \mapsto \beta'']$. Since ord fulfills the law **P1** we have $bn'$ ord $bn'$ for all $bn'$; so in particular,

$$bn[\beta \mapsto \beta''] \text{ ord } (bn[\beta \mapsto \beta'])[\beta' \mapsto \beta''].$$

From the definition of $\mathsf{ord}_p$ we then get

$$\text{REC}\beta.bn \text{ ord}_p \text{ REC}\beta'.bn[\beta \mapsto \beta'],$$

and (1) follows from the definition of ord and $\mathsf{ord}_s$. The proof of (2) is similar. □