# JOIN($X$): Constraint-Based Type Inference for the Join-Calculus

Sylvain Conchon and François Pottier

INRIA Rocquencourt, {Sylvain.Conchon,Francois.Pottier}@inria.fr

**Abstract.** We present a generic constraint-based type system for the join-calculus. The key issue is type generalization, which, in the presence of concurrency, must be restricted. We first define a liberal generalization criterion, and prove it correct. Then, we find that it hinders type inference, and propose a cruder one, reminiscent of ML's *value restriction*.
We establish type safety using a *semi-syntactic* technique, which we believe is of independent interest. It consists in interpreting typing judgements as (sets of) judgements in an underlying system, which itself is given a syntactic soundness proof. This hybrid approach allows giving pleasant logical meaning to high-level notions such as type variables, constraints and generalization, and clearly separating them from low-level aspects (substitution lemmas, etc.), which are dealt with in a simple, standard way.

## 1 Introduction

The join-calculus [2] is a name-passing process calculus related to the asynchronous $\pi$-calculus. The original motivation for its introduction was to define a process calculus amenable to a distributed implementation. In particular, the join-calculus merges reception, restriction and replication into a single syntactic form, the `def` construct, avoiding the need for distributed consensus. This design decision turns out to also have an important impact on typing. Indeed, because the behavior of a channel is fully known at definition time, its type can be safely generalized. Thus, `def` constructs become analogous to ML's `let` definitions. For instance, the following definition:

```
def apply(f,x) = f(x)
```

defines a channel `apply` which expects two arguments `f` and `x` and, upon receipt, sends the message `f(x)`. In Fournet *et al.*'s type system [3], `apply` receives the parametric type scheme $\forall\alpha.\langle\langle\alpha\rangle,\alpha\rangle$, where $\langle\cdot\rangle$ is the channel type constructor.

### 1.1 Motivation

Why develop a new type system for the join-calculus? The unification-based system proposed by Fournet *et al.* [3] shares many attractive features with ML's type system: it is simple, expressive, and easy to implement, as shown by the JoCaml experiment [1]. Like ML, it is *prescriptive*, i.e. intended to infer reasonably simple types and to enforce a programming discipline.

Type systems are often used as a nice formal basis for various program analyses, such as control flow analysis, strictness analysis, usage analysis, and so on. These systems, however, tend to be essentially *descriptive*, i.e. intended to infer accurate types and to reject as few programs as possible. To achieve this goal, it is common to describe the behavior of programs using a rich *constraint* language, possibly involving subtyping, set constraints, conditional constraints, etc. We wish to define such a descriptive type system for the join-calculus, as a vehicle for future type-based analyses.

Following Odersky *et al.* [5], we parameterize our type system with an arbitrary constraint logic $X$, making it more generic and more easily re-useable. Our work may be viewed as an attempt to adapt their constraint-based framework to the join-calculus, much as Fournet *et al.* adapted ML's type discipline.

## 1.2 Type Generalization Criteria

The `def` construct improves on `let` expressions by allowing synchronization between channels. Thus, we can define a variant of `apply` that receives the channel `f` and the argument `x` from different channels.

```
  def apply(f) | args(x) = f(x)
```

This simultaneously defines the names `apply` and `args`. The message `f(x)` will be emitted whenever a message is received on both of these channels.

In a subtyping-constraint-based type system, one would expect `apply` and `args` to be given types $\langle\beta\rangle$ and $\langle\alpha\rangle$, respectively, *correlated* by the constraint $\beta \leq \langle\alpha\rangle$. The constraint requires the channels to be used in a *consistent* way: the type of `x` must match the expectations of `f`. Now, if we were to generalize these types separately, we would obtain `apply` : $\forall\alpha\beta[\beta \leq \langle\alpha\rangle].\langle\beta\rangle$ and `args` : $\forall\alpha\beta[\beta \leq \langle\alpha\rangle].\langle\alpha\rangle$, which are logically equivalent to `apply` : $\forall\alpha.\langle\langle\alpha\rangle\rangle$ and `args` : $\forall\alpha.\langle\alpha\rangle$. These types no longer reflect the consistency requirement!

To address this problem, Fournet *et al.* state that any type variable which is *shared* between two jointly defined names (here, `apply` and `args`), i.e. which occurs free in their types, must not be generalized. However, this criterion is based on the *syntax* of types, and makes little sense in the presence of an arbitrary constraint logic $X$. In the example above, `apply` and `args` have types $\langle\beta\rangle$ and $\langle\alpha\rangle$, so they share no type variables. The correlation is only apparent in the constraint $\beta \leq \langle\alpha\rangle$. When the constraint logic $X$ is known, correlations can be detected by examining the (syntax of the) constraint, looking for *paths* connecting $\alpha$ and $\beta$. However, we want our type system to be parametric in $X$, so the syntax (and the meaning) of constraints is, in general, not available. This leads us to define a uniform, *logical* generalization criterion (Sect. 5.2), which we prove sound.

Unfortunately, and somewhat surprisingly, this criterion turns out to hinder type inference. As a result, we will propose a cruder one, reminiscent of ML's so-called *value restriction* [9].

$$\begin{aligned}
\texttt{def}\ D, J \triangleright P\ \texttt{in}\ Q &\rightleftharpoons \texttt{def}\ D, \varphi J \triangleright \varphi P\ \texttt{in}\ Q \\
&\qquad \text{if } \mathrm{dom}(\varphi) = \mathrm{ln}(J) \wedge \mathrm{codom}(\varphi) \cap \mathrm{fn}(J \triangleright P) = \varnothing \\
\texttt{def}\ D\ \texttt{in}\ P &\rightleftharpoons \texttt{def}\ \varphi D\ \texttt{in}\ \varphi P \\
&\qquad \text{if } \mathrm{dom}(\varphi) = \mathrm{dn}(D) \wedge \mathrm{codom}(\varphi) \cap \mathrm{fn}(\texttt{def}\ D\ \texttt{in}\ P) = \varnothing
\end{aligned}$$

$$\begin{array}{rclcrcl}
P \mid Q &\rightleftharpoons& Q \mid P & \qquad & D_1, D_2 &\rightleftharpoons& D_2, D_1 \\
P \mid 0 &\rightleftharpoons& P & & D, \epsilon &\rightleftharpoons& D \\
P \mid (Q \mid R) &\rightleftharpoons& (P \mid Q) \mid R & & D_1, (D_2, D_3) &\rightleftharpoons& (D_1, D_2), D_3
\end{array}$$

$$\begin{aligned}
P \rightarrow Q &\Rightarrow P \mid R \rightarrow Q \mid R \\
P \rightarrow Q &\Rightarrow \texttt{def}\ D\ \texttt{in}\ P \rightarrow \texttt{def}\ D\ \texttt{in}\ Q
\end{aligned}$$

$$\begin{array}{rcll}
(\texttt{def}\ D\ \texttt{in}\ P) \mid Q &\rightleftharpoons& \texttt{def}\ D\ \texttt{in}\ (P \mid Q) & \text{if } \mathrm{dn}(D) \cap \mathrm{fn}(Q) = \varnothing \\
\texttt{def}\ D_1\ \texttt{in}\ \texttt{def}\ D_2\ \texttt{in}\ P &\rightleftharpoons& \texttt{def}\ D_1, D_2\ \texttt{in}\ P & \text{if } \mathrm{fn}(D_1) \cap \mathrm{dn}(D_2) = \varnothing \\
\texttt{def}\ D, J \triangleright P\ \texttt{in}\ Q \mid \varphi J &\rightarrow& \texttt{def}\ D, J \triangleright P\ \texttt{in}\ Q \mid \varphi P & \text{if } \mathrm{dom}(\varphi) = \mathrm{ln}(J)
\end{array}$$

**Fig. 1.** Operational semantics

### 1.3 Overview

We first recall the syntax and semantics of the join-calculus, and introduce some useful notation. Then, we introduce a *ground* type system for the join-calculus, called B($T$), and establish its correctness in a syntactic way (Sect. 4). Building on this foundation, Sect. 5 introduces JOIN($X$) and proves it correct with respect to B($T$). Sect. 6 studies type reconstruction, suggesting that a restricted generalization criterion must be adopted in order to obtain a complete algorithm.

## 2 The Join-Calculus

We assume given a countable set of names $\mathcal{N}$, ranged over by $x, y, u, v, \ldots$ We write $\vec{u}$ for a tuple $(u_1, \ldots, u_n)$ and $\bar{u}$ for a set $\{u_1, \ldots, u_n\}$ , where $n \geq 0$. The syntax of the join-calculus is as follows.

$$\begin{aligned} P &::= 0 \mid (P \mid P) \mid u\,\langle \vec{v}\,\rangle \mid \texttt{def}\ D\ \texttt{in}\ P \\ D &::= \epsilon \mid J \triangleright P \mid D, D \\ J &::= u\,\langle \vec{y}\,\rangle \mid (J \mid J) \end{aligned}$$

We require join patterns to be linear. That is, all defined names in a join pattern must be pairwise different, and all local names must also be pairwise different. The set of *defined names* $\mathrm{dn}(J)$ and the set of *local names* $\mathrm{ln}(J)$ of a join-pattern $J$ are defined as follows.

$$\begin{aligned} \mathrm{dn}(u\,\langle \vec{y}\,\rangle) &= \{u\} & \qquad \mathrm{dn}(J \mid J') &= \mathrm{dn}(J) \cup \mathrm{dn}(J') \\ \mathrm{ln}(u\,\langle \vec{y}\,\rangle) &= \bar{y} & \qquad \mathrm{ln}(J \mid J') &= \mathrm{ln}(J) \cup \mathrm{ln}(J') \end{aligned}$$

The set of defined names $\mathrm{dn}(D)$ of a definition $D$ is the union of the sets $\mathrm{dn}(J)$ of all join-patterns $J$ which appear in $D$. Then, the set of *free names* of a process or of a definition are as follows.

$$\begin{aligned} \mathrm{fn}(0) &= \varnothing & \qquad \mathrm{fn}(\epsilon) &= \varnothing \\ \mathrm{fn}(u\,\langle \vec{v}\,\rangle) &= \{u\} \cup \bar{v} & \qquad \mathrm{fn}(J \triangleright P) &= \mathrm{fn}(P) \setminus \mathrm{ln}(J) \\ \mathrm{fn}(P \mid P') &= \mathrm{fn}(P) \cup \mathrm{fn}(P') & \qquad \mathrm{fn}(D, D') &= \mathrm{fn}(D) \cup \mathrm{fn}(D') \\ \mathrm{fn}(\texttt{def}\ D\ \texttt{in}\ P) &= (\mathrm{fn}(D) \cup \mathrm{fn}(P)) \setminus \mathrm{dn}(D) \end{aligned}$$

This defines the scoping rules of the language. The local names of a join-pattern (i.e. the formal parameters of its messages) are bound in the corresponding guarded process, while defined names (i.e. channels being created) are bound within the whole defining process, that is, within all guarded processes as well as within the main process.

*Reduction* $\rightarrow$ is defined as the smallest relation that satisfies the laws in Fig. 1. $\varphi$ ranges over *renamings*, i.e. one-to-one maps from $\mathcal{N}$ into $\mathcal{N}$. $\rightleftharpoons$ stands for $\rightarrow \cap \leftarrow$. It is customary to distinguish structural equivalence and reduction, but this is unnecessary here.

## 3 Notation

This section defines some mathematical notation used throughout the paper.

**Definition 3.1.** *Given a set $T$, a $T$-*environment*, usually denoted $\Gamma$, is a partial mapping from $\mathcal{N}$ into $T$. If $N \subseteq \mathcal{N}$, $\Gamma|_N$ denotes the restriction of $\Gamma$ to $N$. $\Gamma + \Gamma'$ is the environment which maps every $u \in \mathcal{N}$ to $\Gamma'(u)$, if it is defined, and to $\Gamma(u)$ otherwise. When $\Gamma$ and $\Gamma'$ agree on $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma')$, $\Gamma + \Gamma'$ is written $\Gamma \oplus \Gamma'$. If $T$ is equipped with a partial order, it is extended point-wise to $T$-environments of identical domain.*

**Definition 3.2.** *Given a set $T$, ranged over by $t$, $\vec{t}$ denotes a tuple $(t_1, \ldots t_n)$, of length $n \geq 0$; we let $T^\star$ denote the set of such tuples. If $T$ is equipped with a partial order, it is extended point-wise to tuples of identical length.*

**Definition 3.3.** *Given a set $I$, $(x_i : t_i)^{i \in I}$ denotes the partial mapping $x_i \mapsto t_i$ of domain $\bar{x} = \{x_i \,;\, i \in I\}$. $(P_i)^{i \in I}$ denotes the parallel composition of the processes $P_i$. $(D_i)^{i \in I}$ denotes the conjunction of the definitions $D_i$.*

**Definition 3.4.** *The* Cartesian product *of a labelled tuple of sets $\mathcal{A} = (x_i : s_i)^{i \in I}$, written $\Pi\mathcal{A}$, is the set of tuples $\{(x_i : t_i)^{i \in I} \,;\, \forall i \in I \quad t_i \in s_i\}$.*

**Definition 3.5.** *Given a partially ordered set $T$ and a subset $V$ of $T$, the* cone *generated by $V$ within $T$, denoted by $\uparrow V$, is $\{t \in T \,;\, \exists v \in V \quad v \leq t\}$. $V$ is said to be* upward-closed *if and only if $V = \uparrow V$.*

## 4 The System B($T$)

This section defines an intermediate type system for the join-calculus, called B($T$). It is a *ground* type system: it does not have a notion of type variable. Instead, it has *monotypes*, taken to be elements of some set $T$, and *polytypes*, merely defined as certain subsets of $T$.

**Assumptions.** We assume given a set $T$, whose elements, usually denoted by $t$, are called *monotypes.* $T$ must be equipped with a partial order $\leq$. We assume given a total function, denoted $\langle \cdot \rangle$, from $T^\star$ into $T$, such that $\langle \vec{t} \rangle \leq \langle \vec{t'} \rangle$ holds if and only if $\vec{t'} \leq \vec{t}$.

**Definition 4.1.** *A* polytype*, usually denoted by $s$, is a non-empty, upward-closed subset of $T$. Let $S$ be the set of all polytypes. We order $S$ by $\supseteq$, i.e. we write $s \leq s'$ if and only if $s \supseteq s'$.*

Note that $\leq$ and $\langle \cdot \rangle$ operate on $T$. Furthermore, $S$ is defined on top of $T$; there is no way to inject $S$ back into $T$. In other words, this presentation allows *rank-1* polymorphism only; impredicative polymorphism is ruled out. This is in keeping with the Hindley-Milner family of type systems [4, 5].

**Definition 4.2.** *A* monotype environment*, denoted by $\mathcal{B}$, is a $T$-environment. A* polytype environment*, denoted by $\Gamma$ or $\mathcal{A}$, is an $S$-environment.*

**Definition 4.3.** *The type system $B(T)$ is given in Fig. 2. By abuse of notation, in the first premise of rule* B-Join*, a monotype binding $(u : t)$ is implicitly viewed as the polytype binding $(u : \uparrow\{t\})$.*

Every typing judgement carries a polytype environment $\Gamma$ on its left-hand side, representing a set of assumptions under which its right-hand side may be used. Right-hand sides come in four varieties. $u : t$ states that the name $u$ has type $t$. $D :: \mathcal{B}$ (resp. $D :: \mathcal{A}$) states that the definition $D$ gives rise to the environment fragment $\mathcal{B}$ (resp. $\mathcal{A}$). Then, dom($\mathcal{B}$) (resp. dom($\mathcal{A}$)) is, by construction, dn($D$). Lastly, a right-hand side of the form $P$ simply states that the process $P$ is well-typed.

The most salient aspect of these rules is their treatment of polymorphism. Rule B-Inst performs *instantiation* by allowing a polytype $s$ to be specialized to

**Names**

$$\textsc{b-Inst}\quad \frac{\Gamma(u) = s \qquad t \in s}{\Gamma \vdash u : t} \qquad\qquad \textsc{b-Sub-Name}\quad \frac{\Gamma \vdash u : t' \qquad t' \leq t}{\Gamma \vdash u : t}$$

**Definitions**

$$\textsc{b-Empty}\quad \Gamma \vdash \epsilon :: \vec{0} \qquad \textsc{b-Join}\quad \frac{\Gamma + (\vec{u}_i : \vec{t}_i)^{i \in I} \vdash P}{\Gamma \vdash (x_i \langle \vec{u}_i \rangle)^{i \in I} \triangleright P :: (x_i : \langle \vec{t}_i \rangle)^{i \in I}} \qquad \textsc{b-Or}\quad \frac{\Gamma \vdash D_1 :: \mathcal{B}_1 \qquad \Gamma \vdash D_2 :: \mathcal{B}_2}{\Gamma \vdash D_1, D_2 :: \mathcal{B}_1 \oplus \mathcal{B}_2}$$

$$\textsc{b-Sub-Def}\quad \frac{\Gamma \vdash D :: \mathcal{B} \qquad \mathcal{B} \leq \mathcal{B}'}{\Gamma \vdash D :: \mathcal{B}'} \qquad\qquad \textsc{b-Gen}\quad \frac{\forall \mathcal{B} \in \Pi\mathcal{A} \quad \Gamma \vdash D :: \mathcal{B}}{\Gamma \vdash D :: \mathcal{A}}$$

**Processes**

$$\textsc{b-Null}\quad \Gamma \vdash 0 \qquad \textsc{b-Par}\quad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \textsc{b-Msg}\quad \frac{\Gamma \vdash u : \langle \vec{t} \rangle \qquad \Gamma \vdash \vec{v} : \vec{t}}{\Gamma \vdash u \langle \vec{v} \rangle}$$

$$\textsc{b-Def}\quad \frac{\Gamma + \mathcal{A} \vdash D :: \mathcal{A} \qquad \Gamma + \mathcal{A} \vdash P}{\Gamma \vdash \texttt{def}\ D\ \texttt{in}\ P}$$

**Fig. 2.** The system B($T$)

any monotype $t \in s$. Conversely, rule b-Gen performs *generalization* by allowing the judgement $\Gamma \vdash D :: (x_i : s_i)^{i \in I}$ to be formed if $\Gamma \vdash D :: (x_i : t_i)^{i \in I}$ holds whenever $(x_i : t_i)^{i \in I} \in \Pi(x_i : s_i)^{i \in I}$, i.e. whenever $\forall i \in I \quad t_i \in s_i$ holds. In other words, this system offers an *extensional* view of polymorphism: a polytype $s$ is definitionally equal to the set of its monotype instances. Through b-Gen and b-Inst, the judgement $\Gamma \vdash D :: \mathcal{A}$ may be viewed as mere evidence for the set of its instances. Note that there may be an infinite number of them, so rule b-Gen may require an infinite number of premises.

The first premise of rule b-Def reflects the fact that every definition is recursive by requiring it to produce an environment fragment $\mathcal{A}$ under assumptions $\Gamma + \mathcal{A}$ (rather than $\Gamma$ alone). The rule expects a polytype environment fragment $\mathcal{A}$ (rather than a monotype environment fragment $\mathcal{B}$), so the system has *polymorphic recursion.*

These remarks show that typechecking in B($T$) is not necessarily decidable. This will not hinder us, since we use B($T$) only as an intermediate step in the construction of a decidable type system, namely JOIN($X$).

Rules other than b-Gen, b-Inst and b-Def are fairly straightforward; they involve monotypes only, and are similar to those found in common typed process calculi. The only non-syntax-directed rules are the subtyping rules, namely b-

SUB-NAME and B-SUB-DEF. Rule B-GEN must (and can only) be applied once above every use of B-DEF, so it is not a source of non-determinism.

Before establishing type soundness for $\mathrm{B}(T)$, we prove several auxiliary lemmas.

**Lemma 4.4 (Depth strengthening).** *If $\Gamma \vdash P$ and $\Gamma' \leq \Gamma$, then $\Gamma' \vdash P$.*

**Lemma 4.5 (Width strengthening).** *If $u \notin \mathrm{fn}(P) \cup \mathrm{fn}(D)$, then*

$$\Gamma \vdash P \Leftrightarrow \Gamma + (u : s) \vdash P$$
$$\Gamma \vdash D :: \mathcal{B} \Leftrightarrow \Gamma + (u : s) \vdash D :: \mathcal{B}$$

**Lemma 4.6 (Substitution).** *If $\Gamma + (u : t) \vdash P$ and $\Gamma \vdash v : t$, then $\Gamma \vdash P[v/u]$.*

**Lemma 4.7.** *A derivation of $\Gamma \vdash u : t$ is* canonical *if and only if it contains no instance of rule* B-SUB-NAME. *Every judgement of the form $\Gamma \vdash u : t$ has a canonical derivation.*

*Proof.* Straightforward consequence of the fact that every polytype is upward-closed.

**Lemma 4.8.** *Every judgement of the form $\Gamma \vdash D_1, D_2 :: \mathcal{B}$ has a derivation which ends with an instance of rule* B-OR. *Every judgement of the form $\Gamma \vdash J \triangleright P :: (x_i : \langle \vec{t_i} \rangle)^{i \in I}$ has a derivation which ends with an instance of rule* B-JOIN.

*Proof.* By induction on the derivation of $\Gamma \vdash D :: \mathcal{B}$. Assume the derivation ends with an instance of B-SUB-DEF:

$$\frac{\Gamma \vdash D :: \mathcal{B}' \qquad \mathcal{B}' \leq \mathcal{B}}{\Gamma \vdash D :: \mathcal{B}}$$

Thanks to the induction hypothesis, we may assume that the premise $\Gamma \vdash D :: \mathcal{B}'$ is itself a consequence of B-JOIN or B-OR.

Case B-JOIN. The rule must be

$$\frac{\Gamma + (\vec{u}_i : \vec{t'}_i)^{i \in I} \vdash P}{\Gamma \vdash J \triangleright P : (x_i : \langle \vec{t'}_i \rangle)^{i \in I}}$$

where $\mathcal{B}' = (x_i : \langle \vec{t'}_i \rangle)^{i \in I}$. Furthermore, by hypothesis, $\mathcal{B}$ is $(x_i : \langle \vec{t_i} \rangle)^{i \in I}$. Considering $\mathcal{B}' \leq \mathcal{B}$, this entails $\forall i \in I \quad \vec{t_i} \leq \vec{t'}_i$. By Lemma 4.4, we then have $\Gamma + (\vec{u}_i : \vec{t_i})^{i \in I} \vdash P$. Rule B-JOIN allows concluding that $\Gamma \vdash D :: \mathcal{B}$.

Case B-OR. Then, $\mathcal{B}'$ may be written $\mathcal{B}'_1 \oplus \mathcal{B}'_2$. The rule is

$$\frac{\Gamma \vdash D_1 :: \mathcal{B}'_1 \qquad \Gamma \vdash D_2 :: \mathcal{B}'_2}{\Gamma \vdash D :: \mathcal{B}'_1 \oplus \mathcal{B}'_2}$$

Considering $\mathcal{B}' \leq \mathcal{B}$, $\mathcal{B}$ must be of the form $\mathcal{B}_1 \oplus \mathcal{B}_2$, where $\mathcal{B}'_1 \leq \mathcal{B}_1$ and $\mathcal{B}'_2 \leq \mathcal{B}_2$. We may then build the derivations

$$\frac{\vdash D_i :: \mathcal{B}'_i \qquad \mathcal{B}'_i \leq \mathcal{B}_i}{\vdash D_i :: \mathcal{B}_i}$$

and conclude by applying rule B-OR.

This tiny lemma will be useful in Sect. 5. It is a corollary of Lemma 4.8.

**Lemma 4.9.** *Assume* $\Gamma \vdash (D, J \triangleright P) :: \mathcal{B}'$ *and* $\mathcal{B}'|_{\mathrm{dn}(J)} \leq \mathcal{B}$. *Then* $\Gamma \vdash J \triangleright P :: \mathcal{B}$.

*Proof.* By Lemma 4.8, some derivation of $\Gamma \vdash (D, J \triangleright P) :: \mathcal{B}'$ must end with

$$\text{B-OR}\ \frac{\ldots \qquad \Gamma \vdash J \triangleright P :: \mathcal{B}'|_{\mathrm{dn}(J)}}{\Gamma \vdash D, J \triangleright P :: \mathcal{B}'}$$

We conclude by building

$$\text{B-SUB-DEF}\ \frac{\Gamma \vdash J \triangleright P :: \mathcal{B}'|_{\mathrm{dn}(J)} \qquad \mathcal{B}'|_{\mathrm{dn}(J)} \leq \mathcal{B}}{\Gamma \vdash J \triangleright P :: \mathcal{B}}$$

We establish type soundness for $\mathrm{B}(T)$ following the syntactic approach of Wright and Felleisen [10], i.e. by proving that $\mathrm{B}(T)$ enjoys *subject reduction* and *progress* properties. Due to the complex syntactic structure of the join-calculus (rather than expressions alone, one must deal with names, join-patterns, definitions and processes), the proof is not particularly short. However, thanks to our abstract treatment of polymorphism, it is entirely straightforward.

Authors of previous type systems for the join-calculus [3, 6] have found that structural equivalence, as presented in Fig. 1, does not preserve typings. Indeed, the last structural equivalence rule, by turning a series of nested definitions into a single, mutually recursive definition, may cause a process to become ill-typed, unless the type system has polymorphic recursion. Because polymorphic recursion leads to undecidability, and because these authors were committed to the syntactic approach to type soundness, they had to restrict structural equivalence and to introduce reduction contexts in the operational semantics. The problem does not arise in this paper, because $\mathrm{B}(T)$ has polymorphic recursion.

**Theorem 4.10 (Subject reduction).** $\Gamma \vdash P$ *and* $P \rightarrow P'$ *imply* $\Gamma \vdash P'$.

*Proof.* We check that each of the rules in Fig. 1 preserves typings. Then, a simple context lemma, which we do not state here, allows concluding that all reductions preserve typings.

$\alpha$-conversion is easily dealt with using an auxiliary renaming lemma. Parallel composition of processes offers no difficulty. Neither does composition of definitions, because we can restrict our attention, without loss of generality, to typings of the form $\Gamma \vdash D :: \mathcal{B}$, and because Lemma 4.8 is available. We now deal with the three remaining cases.

*Scope extrusion.* The processes at hand are `def` $D$ `in` $P \mid Q$ and `def` $D$ `in` $P \mid Q$, where $\mathrm{dn}(D) \cap \mathrm{fn}(Q) = \varnothing$. Considering rules B-PAR and B-DEF, these processes are well-typed within $\Gamma$ if and only if there exists $\mathcal{A}$ such that $\Gamma + \mathcal{A} \vdash D :: \mathcal{A}$, $\Gamma + \mathcal{A} \vdash P$ and $\Gamma \vdash Q$ (resp. $\Gamma + \mathcal{A} \vdash Q$). Because $\mathrm{dom}(\mathcal{A})$ must be $\mathrm{dn}(D)$, and because $\mathrm{dn}(D) \cap \mathrm{fn}(Q) = \varnothing$, Lemma 4.5 shows that $\Gamma \vdash Q$ holds if and only if $\Gamma + \mathcal{A} \vdash Q$ holds. The result follows.

*Merging of definitions.* The processes at hand are `def` $D_1$ `in def` $D_2$ `in` $P$ and `def` $D_1, D_2$ `in` $P$, where $\mathrm{fn}(D_1) \cap \mathrm{dn}(D_2) = \varnothing$. Furthermore, we may assume, without loss of generality, that $\mathrm{dn}(D_1) \cap \mathrm{dn}(D_2) = \varnothing$. (If this is not the case, perform $\alpha$-conversion first.) Considering rules B-PAR and B-DEF, the former is well-typed within $\Gamma$ if and only if there exist $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\Gamma + \mathcal{A}_1 \vdash D_1 :: \mathcal{A}_1$, $\Gamma + \mathcal{A}_1 + \mathcal{A}_2 \vdash D_2 :: \mathcal{A}_2$ and $\Gamma + \mathcal{A}_1 + \mathcal{A}_2 \vdash P$, while the latter is well-typed within $\Gamma$ if and only if there exists $\mathcal{A}$ such that $\Gamma + \mathcal{A} \vdash D_1, D_2 :: \mathcal{A}$ and $\Gamma + \mathcal{A} \vdash P$. There remains to show that these conditions are equivalent.

We begin by noticing that $\Gamma + \mathcal{A}_1 \vdash D_1 :: \mathcal{A}_1$ is equivalent to $\Gamma + \mathcal{A}_1 + \mathcal{A}_2 \vdash D_1 :: \mathcal{A}_1$. Indeed, this follows from Lemma 4.5, using $\mathrm{fn}(D_1) \cap \mathrm{dn}(D_2) = \varnothing$ and $\mathrm{dom}(\mathcal{A}_2) = \mathrm{dn}(D_2)$.

Now, assume the first condition holds. Define $\mathcal{A} = \mathcal{A}_1 \oplus \mathcal{A}_2$. Assume $\mathcal{B} \in \Pi\mathcal{A}$. $\mathcal{B}$ may be written $\mathcal{B}_1 \oplus \mathcal{B}_2$, where $\mathcal{B}_i \in \Pi\mathcal{A}_i$ for $i \in \{1, 2\}$. We have $\Gamma + \mathcal{A} \vdash D_i :: \mathcal{A}_i$ for $i \in \{1, 2\}$. The derivations of these judgements must end with

$$\text{B-GEN}\ \frac{\forall \mathcal{B} \in \Pi\mathcal{A}_i \quad \Gamma + \mathcal{A} \vdash D_i :: \mathcal{B}}{\Gamma + \mathcal{A} \vdash D_i :: \mathcal{A}_i}$$

Thus, we have $\Gamma + \mathcal{A} \vdash D_i :: \mathcal{B}_i$ for $i \in \{1, 2\}$. By rule B-OR, $\Gamma + \mathcal{A} \vdash D_1, D_2 :: \mathcal{B}$ holds. Recalling that $\mathcal{B}$ was arbitrary, rule B-GEN yields $\Gamma + \mathcal{A} \vdash D_1, D_2 :: \mathcal{A}$. Thus, the second condition holds.

Conversely, assume the second condition holds. Let $\mathcal{A}_i$ stand for the restriction of $\mathcal{A}$ to $\mathrm{dn}(D_i)$, for $i \in \{1, 2\}$. We have $\mathcal{A} = \mathcal{A}_1 \oplus \mathcal{A}_2$. Choose $i$, $j$ such that $\{i, j\} = \{1, 2\}$. Consider some $\mathcal{B}_i \in \Pi\mathcal{A}_i$. Because every polytype is non-empty, there exists some $B_j$ such that $\mathcal{B}_i \oplus \mathcal{B}_j \in \Pi\mathcal{A}$. The derivation of $\Gamma + \mathcal{A} \vdash D_1, D_2 :: \mathcal{A}$ must end with

$$\text{B-GEN}\ \frac{\forall \mathcal{B} \in \Pi\mathcal{A} \quad \Gamma + \mathcal{A} \vdash D_1, D_2 :: \mathcal{B}}{\Gamma + \mathcal{A} \vdash D_1, D_2 :: \mathcal{A}}$$

Among the premises, we find the judgement $\Gamma + \mathcal{A} \vdash D_1, D_2 :: \mathcal{B}_i \oplus \mathcal{B}_j$, some derivation of which must have $\Gamma + \mathcal{A} \vdash D_i :: \mathcal{B}_i$ as one of its premises. Because $\mathcal{B}_i$ was arbitrary, rule B-GEN yields $\Gamma + \mathcal{A} \vdash D_i :: \mathcal{A}_i$. The result follows.

*Reduction.* The processes at hand are `def` $D, J \triangleright P$ `in` $Q \mid \varphi J$ and `def` $D, J \triangleright P$ `in` $Q \mid \varphi P$, where $\mathrm{dom}(\varphi) = \mathrm{ln}(J)$. $J$ must be of the form $(x_i \langle \vec{u}_i \rangle)^{i \in I}$. Then, $\mathrm{dom}(\varphi) = \cup_{i \in I} \bar{u}_i$. Assume $\Gamma \vdash$ `def` $D, J \triangleright P$ `in` $Q \mid \varphi J$. Considering rule B-DEF, there exists some $\mathcal{A}$ such that $\Gamma + \mathcal{A} \vdash D, J \triangleright P :: \mathcal{A}$ and $\Gamma + \mathcal{A} \vdash Q \mid \varphi J$. A derivation of the former must end with

$$\text{B-GEN}\ \frac{\forall \mathcal{B} \in \Pi\mathcal{A} \quad \Gamma + \mathcal{A} \vdash D, J \triangleright P :: \mathcal{B}}{\Gamma + \mathcal{A} \vdash D, J \triangleright P :: \mathcal{A}}$$

The derivation of the latter must include

$$\text{B-PAR}\ \dfrac{\forall i \in I \quad \text{B-MSG}\ \dfrac{\text{B-INST}\ \dfrac{(\Gamma + \mathcal{A})(x_i) = s_i \quad \langle \vec{t_i} \rangle \in s_i}{\Gamma + \mathcal{A} \vdash x_i : \langle \vec{t_i} \rangle} \qquad \Gamma + \mathcal{A} \vdash \varphi \vec{u}_i : \vec{t}_i}{\Gamma + \mathcal{A} \vdash x_i \langle \varphi \vec{u}_i \rangle}}{\Gamma + \mathcal{A} \vdash \varphi J}$$

(By Lemma 4.7, we have assumed a canonical derivation of $\Gamma + \mathcal{A} \vdash x_i : \langle \vec{t_i} \rangle$.) We have $\text{dom}(\mathcal{A}) = \text{dn}(D, J \triangleright P) \supseteq \bar{x}$. Given $(\Gamma + \mathcal{A})(x_i) = s_i$, this means $\mathcal{A}(x_i) = s_i$. Let $\mathcal{B} = (x_i : \langle \vec{t_i} \rangle)^{i \in I}$. Then, some extension $\mathcal{B}'$ of $\mathcal{B}$ is a member of $\Pi\mathcal{A}$, which means $\Gamma + \mathcal{A} \vdash D, J \triangleright P :: \mathcal{B}'$ is among the premises of B-GEN above. By Lemma 4.8, some derivation of it is of the form

$$\text{B-OR}\ \dfrac{\dots \qquad \text{B-JOIN}\ \dfrac{\Gamma + \mathcal{A} + (\vec{u}_i : \vec{t}_i)^{i \in I} \vdash P}{\Gamma + \mathcal{A} \vdash J \triangleright P :: \mathcal{B}}}{\Gamma + \mathcal{A} \vdash D, J \triangleright P :: \mathcal{B}'}$$

Thus, we have established $\Gamma + \mathcal{A} + (\vec{u}_i : \vec{t}_i)^{i \in I} \vdash P$ and $\Gamma + \mathcal{A} \vdash \varphi \vec{u}_i : \vec{t}_i$. Lemma 4.6 yields $\Gamma + \mathcal{A} \vdash P[\varphi \vec{u}_i / \vec{u}_i]$, i.e. $\Gamma + \mathcal{A} \vdash \varphi P$. The result follows.

Our notion of progress is weak: we guarantee the absence of runtime errors (caused by arity mismatches), but we do not prove the absence of deadlocks. This is a common weakness of many type systems for process calculi.

**Definition 4.11.** *A process of the form* `def` $D, J \triangleright P$ `in` $Q \mid u \langle \vec{v} \rangle$ *is* faulty *if* $J$ *defines a message* $u \langle \vec{y} \rangle$ *where* $\vec{v}$ *and* $\vec{y}$ *have different arities.*

Please note that a faulty process is not necessarily irreducible.

**Theorem 4.12 (Progress).** *No well-typed process is faulty.*

*Proof.* Assume $\Gamma \vdash$ `def` $D, J \triangleright P$ `in` $Q \mid u \langle \vec{v} \rangle$ and $J$ contains a message $u \langle \vec{y} \rangle$. According to B-DEF and B-PAR, we have

$$\Gamma + \mathcal{A} \vdash u \langle \vec{v} \rangle$$
$$\Gamma + \mathcal{A} \vdash D, J \triangleright P :: \mathcal{A}$$

for some polytype environment $\mathcal{A}$. The derivation of the former must be

$$\text{B-MSG}\ \dfrac{\text{B-INST}\ \dfrac{(\Gamma + \mathcal{A})(u) = s \qquad \langle \vec{t} \rangle \in s}{\Gamma + \mathcal{A} \vdash u : \langle \vec{t} \rangle} \qquad \Gamma + \mathcal{A} \vdash \vec{v} : \vec{t}}{\Gamma + \mathcal{A} \vdash u \langle \vec{v} \rangle}$$

(By Lemma 4.7, we have assumed a canonical derivation of $\Gamma + \mathcal{A} \vdash u : \langle \vec{t} \rangle$.) Note that $\vec{v}$ and $\vec{t}$ have the same arity. Because $u \in \text{dn}(J \triangleright P) \subseteq \text{dom}(\mathcal{A})$, $(\Gamma + \mathcal{A})(u) = s$ may be read $\mathcal{A}(u) = s$. Thus, considering every polytype is non-empty, there exists $\mathcal{B} \in \Pi\mathcal{A}$ such that $\mathcal{B}(u) = \langle\vec{t}\rangle$. Now, any derivation of $\Gamma + \mathcal{A} \vdash D, J \triangleright P :: \mathcal{A}$ ends with an instance of B-GEN, among whose premises we find $\Gamma + \mathcal{A} \vdash D, J \triangleright P :: \mathcal{B}$. Because $\mathcal{B}(u) = \langle\vec{t}\rangle$, and because $J$ contains a message $u\,\langle\,\vec{y}\,\rangle$, some derivation of it must be of the form

$$\textsc{b-Or}\ \dfrac{\dots \qquad \textsc{b-Sub-Def}\ \dfrac{\textsc{b-Join}\ \dfrac{\Gamma + \ldots + \vec{y} : \vec{t'} \vdash P}{\Gamma + \mathcal{A} \vdash J \triangleright P :: \mathcal{B}'} \qquad \mathcal{B}' \leq \mathcal{B}|_{\mathrm{dn}(J)}}{\Gamma + \mathcal{A} \vdash J \triangleright P :: \mathcal{B}|_{\mathrm{dn}(J)}}}{\Gamma + \mathcal{A} \vdash D, J \triangleright P :: \mathcal{B}}$$

We have $\mathcal{B}'(u) = \langle\vec{t'}\rangle$, $\mathcal{B}' \leq \mathcal{B}|_{\mathrm{dn}(J)}$, and $\mathcal{B}(u) = \langle\vec{t}\rangle$, which imply that $\vec{t'}$ and $\vec{t}$ have the same arity. As a result, $\vec{y}$ and $\vec{t}$ have the same arity. The result follows.

## 5 The System JOIN($X$)

### 5.1 Presentation

Like B($T$), JOIN($X$) is parameterized by a set of ground types $T$, equipped with a type constructor $\langle\cdot\rangle$ and a subtyping relation $\leq$. It is further parameterized by a first-order logic $X$, interpreted in $T$, whose variables and formulas are respectively called *type variables* and *constraints*. The logic allows describing subsets of $T$ as constraints. Provided constraint satisfiability is decidable, this gives rise to a type system where type checking is decidable.

Our treatment is inspired by the framework HM($X$) [5, 8, 7]. Our presentation differs, however, by explicitly viewing constraints as formulas interpreted in $T$, rather than as elements of an abstract cylindric constraint system. This presentation is more concise, and gives us the ability to explicitly manipulate *solutions* of constraints, an essential requirement in our formulation of type soundness (Theorem 5.10). Even though we lose some generality with respect to the cylindric-system approach, we claim the framework remains general enough.

**Assumptions.** We assume given $(T, \leq, \langle\cdot\rangle)$ as in Sect. 4. Furthermore, we assume given a constraint logic $X$ whose syntax *includes* the following productions:

$$C ::= \mathbf{true} \mid \alpha = \langle\vec{\beta}\rangle \mid \alpha \leq \beta \mid C \wedge C \mid \exists\bar{\alpha}.C \mid \ldots$$

($\alpha, \beta, \ldots$ range over a denumerable set of type variables $\mathcal{V}$.) The syntax of constraints is only partially specified; this allows custom constraint forms, not known in this paper, to be later introduced.

The logic $X$ must be equipped with an interpretation in $T$, i.e. a two-place predicate $\vdash$ whose first argument is an assignment, i.e. a total mapping $\rho$ from $\mathcal{V}$ into $T$, and whose second argument is a constraint $C$. The interpretation must be standard, i.e. satisfy the following laws:

$$\begin{array}{rcl} & \rho \vdash \mathbf{true} & \\ \rho \vdash \alpha_0 = \langle \vec{\alpha}_1 \rangle & \text{iff} & \rho(\alpha_0) = \langle \rho(\vec{\alpha}_1) \rangle \\ \rho \vdash \alpha_0 \leq \alpha_1 & \text{iff} & \rho(\alpha_0) \leq \rho(\alpha_1) \\ \rho \vdash C_0 \wedge C_1 & \text{iff} & \rho \vdash C_0 \wedge \rho \vdash C_1 \\ \rho \vdash \exists \bar{\alpha}.C & \text{iff} & \exists \rho' \quad (\rho' \setminus \bar{\alpha} = \rho \setminus \bar{\alpha}) \wedge \rho' \vdash C \end{array}$$

($\rho \setminus \bar{\alpha}$ denotes the restriction of $\rho$ to $\mathcal{V} \setminus \bar{\alpha}$.) The interpretation of any unknown constraint forms is left unspecified. We write $C \Vdash C'$ if and only if $C$ entails $C'$, i.e. if and only if every solution $\rho$ of $C$ satisfies $C'$ as well.

Note that we do not define a syntax of *types*. As pointed out in [8], types are a useful notation in practice, but are entirely superfluous in theory, since their structure can be encoded into constraints. Therefore, we leave the introduction of types as an implementation issue.

JOIN($X$) has *constrained type schemes*, where a number of type variables $\bar{\alpha}$ are universally quantified, subject to a constraint $C$.

**Definition 5.1.** *A* type scheme *is a triple of a set of quantifiers $\bar{\alpha}$, a constraint $C$, and a type variable $\alpha$; we write $\sigma = \forall \bar{\alpha}[C].\alpha$. The type variables in $\bar{\alpha}$ are bound in $\sigma$; type schemes are considered equal modulo $\alpha$-conversion. By abuse of notation, a type variable $\alpha$ may be viewed as a type scheme $\forall \varnothing[\mathbf{true}].\alpha$. The set of type schemes is written $\mathcal{S}$.*

**Definition 5.2.** *A* polymorphic typing environment*, denoted by $\Gamma$ or $A$, is a $\mathcal{S}$-environment. A* monomorphic typing environment*, denoted by $B$, is a $\mathcal{V}$-environment.*

**Definition 5.3.** *JOIN($X$) is defined by Fig. 3. Every judgement $C, \Gamma \vdash \mathcal{J}$ is implicitly accompanied by the side condition that $C$ must be satisfiable.*

JOIN($X$) differs from B($T$) by replacing monotypes with type variables, polytypes with type schemes, and parameterizing every judgement with a constraint $C$, which represents an assumption about its free type variables. Rule Weaken allows strengthening this assumption, while $\exists$ Intro allows hiding auxiliary type variables which appear nowhere but in the assumption itself. These rules, which are common to names, definitions, and processes, allow constraint simplification.

Because we do not have syntax for types, rules Join and Msg use constraints of the form $\beta = \langle \vec{\alpha} \rangle$ to encode type structure into constraints.

Our treatment of constrained polymorphism is standard. Whereas B($T$) takes an extensional view of polymorphism, JOIN($X$) offers the usual, *intensional* view. Type schemes are introduced by rule Def, and eliminated by Inst. Because implicit $\alpha$-conversion is allowed, every instance of Inst is able to rename the bound variables at will.

For the sake of readability, we have simplified rule Def, omitting two features present in HM($X$)'s $\forall$ Intro rule [5]. First, we do not force the introduction of existential quantifiers in the judgement's conclusion. In the presence of Weaken and $\exists$ Intro, doing so would not affect the set of valid typing judgements, so we prefer a simpler rule. Second, we move the whole constraint $C$ into the type schemes $\forall\bar{\alpha}[C]\twoheadrightarrow B$, whereas it would be sufficient to copy only the part of $C$ where $\bar{\alpha}$ actually occurs. This optimization can be easily added back in if desired.

## 5.2 A Look at the Generalization Condition

The most subtle (and, it turns out, questionable; see Sect. 6.1) aspect of this system is the generalization condition, i.e. the third premise of rule Def, which determines which type variables may be safely generalized. We will now describe it in detail. To begin, let us introduce some notation.

**Definition 5.4.** *If $B = (x_i : \beta_i)^{i\in I}$, then $\forall\bar{\alpha}[C]\twoheadrightarrow B$ is the polymorphic environment $(x_i : \forall\bar{\alpha}[C].\beta_i)^{i\in I}$. This must not be confused with the notation $\forall\bar{\alpha}[C].B$, where the universal quantifier lies outside of the environment fragment $B$.*

The existence of these two notations, and the question of whether it is legal to confuse the two, is precisely at the heart of the generalization issue. Let us have a look at rule Def. Its first premise associates a monomorphic environment fragment $B$ to the definition $D = (J_i \triangleright P_i)^{i\in I}$. If the type variables $\bar{\alpha}$ do not appear free in $\Gamma$, then it is surely correct to generalize the fragment as a whole, i.e. to assert that $D$ has type $\forall\bar{\alpha}[C].B$. However, this is no longer a valid environment fragment, because the quantifier appears *in front of* the whole vector; so, we cannot typecheck $P$ under $\Gamma + \forall\bar{\alpha}[C].B$. Instead, we must push the universal quantifier down into *each* binding, yielding $\forall\bar{\alpha}[C]\twoheadrightarrow B$, which is a well-formed environment fragment, and can be used to augment $\Gamma$.

However, $\forall\bar{\alpha}[C]\twoheadrightarrow B$ may be strictly more general than $\forall\bar{\alpha}[C].B$, because it binds $\bar{\alpha}$ separately in each entry, rather than once in common. We must avoid this situation, which would allow inconsistent uses of the defined names, by properly restricting $\bar{\alpha}$. (When $\bar{\alpha}$ is empty, the two notions coincide.)

To ensure that $\forall\bar{\alpha}[C]\twoheadrightarrow B$ and $\forall\bar{\alpha}[C].B$ coincide, previous works [3, 6] propose syntactic criteria, which forbid generalization of a type variable if it appears free in two distinct bindings in $B$. In an arbitrary constraint logic, however, a syntactic occurrence of a type variable does not necessarily constrain its value. So, it seems preferable to define a logical, rather than syntactic, criterion. To do so, we first give logical meaning to the notations $\forall\bar{\alpha}[C]\twoheadrightarrow B$ and $\forall\bar{\alpha}[C].B$.

**Definition 5.5.** *The denotation of a type scheme $\sigma = \forall\bar{\alpha}[C].\alpha$ under an assignment $\rho$, written $[\![\sigma]\!]_\rho$, is defined as $\uparrow\{\rho'(\alpha)\,;\,(\rho'\setminus\bar{\alpha} = \rho\setminus\bar{\alpha}) \wedge \rho' \vdash C\}$ if this set is non-empty; it is undefined otherwise.*

This definition interprets a type scheme $\sigma$ as the set of its instances in $T$, or, more precisely, as the upper cone which they generate. (Taking the cone accounts for the subtyping relationship ambient in $T$.) It is parameterized by an assignment $\rho$, which gives meaning to the free type variables of $\sigma$.

**Names**

$$\textsc{Inst}\quad \frac{\Gamma(u) = \forall\bar\alpha[C].\alpha}{C, \Gamma \vdash u : \alpha} \qquad\qquad \textsc{Sub-Name}\quad \frac{C, \Gamma \vdash u : \alpha' \qquad C \Vdash \alpha' \leq \alpha}{C, \Gamma \vdash u : \alpha}$$

**Definitions**

$$\textsc{Empty}\quad C, \Gamma \vdash \epsilon :: \vec{0} \qquad\qquad \textsc{Join}\quad \frac{C, \Gamma + (\vec{u_i} : \vec{\alpha_i})^{i \in I} \vdash P \qquad \forall i \in I \quad C \Vdash \beta_i = \langle \vec{\alpha_i} \rangle}{C, \Gamma \vdash (x_i \langle\, \vec{u}_i \,\rangle)^{i \in I} \triangleright P :: (x_i : \beta_i)^{i \in I}}$$

$$\textsc{Or}\quad \frac{C, \Gamma \vdash D_1 : B_1 \qquad C, \Gamma \vdash D_2 : B_2}{C, \Gamma \vdash D_1, D_2 :: B_1 \oplus B_2} \qquad\qquad \textsc{Sub-Def}\quad \frac{C, \Gamma \vdash D :: B' \qquad C \Vdash B' \leq B}{C, \Gamma \vdash D :: B}$$

**Processes**

$$\textsc{Null}\quad C, \Gamma \vdash 0 \qquad \textsc{Par}\quad \frac{C, \Gamma \vdash P \qquad C, \Gamma \vdash Q}{C, \Gamma \vdash P \mid Q} \qquad \textsc{Msg}\quad \frac{C, \Gamma \vdash u : \beta \qquad C, \Gamma \vdash \vec{v} : \vec{\alpha} \qquad C \Vdash \beta = \langle \vec{\alpha} \rangle}{C, \Gamma \vdash u \,\langle\, \vec{v} \,\rangle}$$

$$\textsc{Def}\quad \frac{\begin{array}{c} C, \Gamma + B \vdash (J_i \triangleright P_i)^{i \in I} :: B \qquad \bar\alpha \cap \mathrm{fv}(\Gamma) = \varnothing \\ \forall i \in I \quad C \Vdash \forall\bar\alpha[C].B|_{\mathrm{dn}(J_i)} \leq \forall\bar\alpha[C]\twoheadrightarrow B|_{\mathrm{dn}(J_i)} \\ C', \Gamma + \forall\bar\alpha[C]\twoheadrightarrow B \vdash P \qquad C' \Vdash C \end{array}}{C', \Gamma \vdash \mathtt{def}\ (J_i \triangleright P_i)^{i \in I}\ \mathtt{in}\ P}$$

**Common**

$$\textsc{Weaken}\quad \frac{C', \Gamma \vdash \mathcal{J} \qquad C \Vdash C'}{C, \Gamma \vdash \mathcal{J}} \qquad\qquad \exists\ \textsc{Intro}\quad \frac{C, \Gamma \vdash \mathcal{J} \qquad \bar\alpha \cap \mathrm{fv}(\Gamma, \mathcal{J}) = \varnothing}{\exists\bar\alpha.C, \Gamma \vdash \mathcal{J}}$$

**Fig. 3.** The system JOIN($X$) (with a tentative Def rule)

**Definition 5.6.** *The denotation of an environment fragment* $A = (u_i : \sigma_i)^{i \in I}$ *under an assignment* $\rho$*, written* $(\!|A|\!)_\rho$*, is defined as* $\Pi[\![A]\!]_\rho = \Pi(u_i : [\![\sigma_i]\!]_\rho)^{i \in I}$*. The denotation of* $\forall\bar{\alpha}[C].B$ *under an assignment* $\rho$*, written* $(\!|\forall\bar{\alpha}[C].B|\!)_\rho$*, is defined as* $\uparrow\{\rho'(B)\,;\,(\rho' \setminus \bar{\alpha} = \rho \setminus \bar{\alpha}) \wedge \rho' \vdash C\}$.

This definition interprets environment fragments as a whole, rather than point-wise. That is, $(\!|\cdot|\!)_\rho$ maps environment fragments to sets of tuples of monotypes. A polymorphic environment fragment $A$ maps each name $u_i$ to a type scheme $\sigma_i$. The fact that these type schemes are independent of one another is reflected in our interpretation of $A$ as the Cartesian product of their interpretations. On the other hand, $\forall\bar{\alpha}[C].B$ is just a type scheme whose body happens to be a tuple, so we interpret it as (the upper cone generated by) the set of its instances, as in Definition 5.5.

Interpreting the notations $\forall\bar{\alpha}[C]\twoheadrightarrow B$ and $\forall\bar{\alpha}[C].B$ within the same mathematical space allows us to give a logical criterion under which they coincide.

**Definition 5.7.** *By definition,* $C \Vdash \forall\bar{\alpha}[C].B \leq \forall\bar{\alpha}[C]\twoheadrightarrow B$ *holds if and only if, under every assignment* $\rho$ *such that* $\rho \vdash C$*,* $(\!|\forall\bar{\alpha}[C].B|\!)_\rho \supseteq (\!|\forall\bar{\alpha}[C]\twoheadrightarrow B|\!)_\rho$ *holds.*

*Example 5.8.* Let $B = (a : \langle\alpha\rangle;\, b : \langle\langle\beta\rangle\rangle)$, $C = \alpha \leq \beta$, and $\bar{\alpha} = \{\alpha, \beta\}$. (For conciseness, we use $\langle\cdot\rangle$ as a type constructor, even though it isn't one; hopefully the meaning is clear enough.) Then, under any assignment, the denotation of $\forall\bar{\alpha}[C].B$ is

$$\uparrow\{(a : \langle t\rangle;\, b : \langle\langle t'\rangle\rangle)\,;\, t, t' \in T \wedge t \leq t'\}$$

whereas that of $\forall\bar{\alpha}[C]\twoheadrightarrow B$ is

$$\uparrow\{(a : \langle t\rangle;\, b : \langle\langle t'\rangle\rangle)\,;\, t, t' \in T\}$$

The former is a strict subset of the latter. (If $\top$ stands for the 0-ary channel type $\langle\rangle$, then $(a : \langle\top\rangle;\, b : \langle\langle\langle\top\rangle\rangle\rangle)$ witnesses this fact.) This shows that the environment fragment $B$, under the constraint $C$, *correlates* the names $a$ and $b$. In other words, these names cannot be used independently at arbitrary types. Indeed, this result is in accordance with the intuitive reading of the type scheme $\forall\alpha\beta[\alpha \leq \beta].(a : \langle\alpha\rangle;\, b : \langle\langle\beta\rangle\rangle)$, namely: "the value sent to $a$ may be sent to whichever channel is sent to $b$".

The strength of this criterion is to be independent of the constraint logic $X$. This allows us to prove JOIN($X$) correct in a pleasant generic way (see Sect. 5.3).

As a final remark, let us point out that, independently of how to *define* the generalization criterion, there is also a question of how to *apply* it. It would be correct for rule DEF to require $C \Vdash \forall\bar{\alpha}[C].B \leq \forall\bar{\alpha}[C]\twoheadrightarrow B$, as in [3]. However, when executing the program, only one clause of the definition at a time will be reduced, so it is sufficient to separately ensure that the messages which appear in each clause have consistent types. As a result, we successively apply the criterion to each clause $J_i \triangleright P_i$, by restricting $B$ to the set of its defined names, yielding $B|_{\mathrm{dn}(J_i)}$. In this respect, we closely follow the JoCaml implementation [1] as well as Odersky *et al.* [6].

### 5.3 Type Soundness, Semi-Syntactically

This section gives a type soundness proof for JOIN($X$) by showing that it is safe with respect to B($T$). That is, we show that every judgement $C, \Gamma \vdash \mathcal{J}$ describes the set of all B($T$) judgements of the form $\rho(\Gamma \vdash \mathcal{J})$, where $\rho \vdash C$. Thus, we give logical (rather than syntactic) meaning to JOIN($X$) judgements, yielding a concise and natural proof. As a whole, the approach is still semi-syntactic, because B($T$) itself has been proven correct in a syntactic way.

We first define some notation.

**Definition 5.9.** *When defined (cf. Definition 5.5), $[\![\sigma]\!]_\rho$ is a polytype, i.e. an element of $S$. The denotation function $[\![\cdot]\!]_\rho$ is extended point-wise to typing environments. As a result, if $\Gamma$ is an $\mathcal{S}$-environment, then $[\![\Gamma]\!]_\rho$ is an $S$-environment.*

This allows us to state the main soundness theorem.

**Theorem 5.10 (Soundness).** *Let $\rho(u : \alpha)$, $\rho(D :: B)$, $\rho(P)$ stand for $u : \rho(\alpha)$, $D :: \rho(B)$, $P$, respectively. Then, $\rho \vdash C$ and $C, \Gamma \vdash \mathcal{J}$ imply $[\![\Gamma]\!]_\rho \vdash \rho(\mathcal{J})$.*

*Proof.* By structural induction on the derivation of the input judgement. We use exactly the notations of Fig. 3. In each case, we assume given some solution $\rho$ of the constraint which appears in the judgement's conclusion.

Case Inst. We have $[\![\Gamma]\!]_\rho(u) = [\![\forall\bar{\alpha}[C].\alpha]\!]_\rho \ni \rho(\alpha)$ because $\rho \vdash C$. The result follows by b-Inst.

Case Sub-Name. The induction hypothesis yields $[\![\Gamma]\!]_\rho \vdash u : \rho(\alpha')$. The second premise implies $\rho(\alpha') \leq \rho(\alpha)$. Apply b-Sub-Name to conclude.

Case Empty. Immediate.

Case Join. Let $B = (x_i : \beta_i)^{i \in I}$. Applying the induction hypothesis to the first premise yields $[\![\Gamma]\!]_\rho + (\vec{u_i} : [\![\vec{\alpha_i}]\!]_\rho)^{i \in I} \vdash P$. Since $[\![\alpha]\!]_\rho$ is $\uparrow\{\rho(\alpha)\}$, this may be written $[\![\Gamma]\!]_\rho + (\vec{u_i} : \rho(\vec{\alpha_i}))^{i \in I} \vdash P$. (Recall the abuse of notation introduced in Definition 4.3.) The second premise implies $\forall i \in I \quad \rho(\beta_i) = \langle \rho(\vec{\alpha_i}) \rangle$. As a result, by b-Join, $[\![\Gamma]\!]_\rho \vdash D : \rho(B)$ holds.

Case Or. Then, $D$ is $D_1 \wedge D_2$ and $B$ is $B_1 \oplus B_2$. Applying the induction hypothesis to the premises yields $[\![\Gamma]\!]_\rho \vdash D_i : \rho(B_i)$. Apply b-Or to conclude.

Case Sub-Def. The induction hypothesis yields $[\![\Gamma]\!]_\rho \vdash D : \rho(B')$. The second premise implies $\rho(B') \leq \rho(B)$. Apply b-Sub-Def to conclude.

Cases Null, Par. Immediate.

Case Msg. Applying the induction hypothesis to the first two premises yields $[\![\Gamma]\!]_\rho \vdash u : \rho(\beta)$ and $[\![\Gamma]\!]_\rho \vdash \vec{v} : \rho(\vec{\alpha})$. The last premise entails $\rho(\beta) = \langle \rho(\vec{\alpha}) \rangle$. Apply b-Msg to conclude.

Case Def. By hypothesis, $\rho \vdash C'$; according to the last premise, $\rho \vdash C$ also holds. Let $A = \forall\bar{\alpha}[C] \twoheadrightarrow B$. Take $\mathcal{B} \in (\!|A|\!)_\rho$. Take $i \in I$ and define $\mathcal{B}_i = \mathcal{B}|_{\mathrm{dn}(J_i)}$. Then, $\mathcal{B}_i$ is a member of $(\!|\forall\bar{\alpha}[C] \twoheadrightarrow B|_{\mathrm{dn}(J_i)}|\!)_\rho$, which, according to the third premise, is a subset of $(\!|\forall\bar{\alpha}[C].B|_{\mathrm{dn}(J_i)}|\!)_\rho$. Thus, there exists an assignment $\rho'$ such that $(\rho' \setminus \bar{\alpha} = \rho \setminus \bar{\alpha}) \wedge \rho' \vdash C$ and $\rho'(B|_{\mathrm{dn}(J_i)}) \leq \mathcal{B}_i$. The induction hypothesis, applied to the first premise and to $\rho'$, yields $[\![\Gamma + B]\!]_{\rho'} \vdash D :: \rho'(B)$. By Lemma 4.9, this implies $[\![\Gamma + B]\!]_{\rho'} \vdash J_i \triangleright P_i :: \mathcal{B}_i$.

Now, because $\bar{\alpha} \cap \mathrm{fv}(\Gamma) = \varnothing$, $[\![\Gamma]\!]_{\rho'}$ is $[\![\Gamma]\!]_{\rho}$. Furthermore, given the properties of $\rho'$, we have $[\![B]\!]_{\rho'} \geq [\![\forall\bar{\alpha}[C]\twoheadrightarrow B]\!]_{\rho} = [\![A]\!]_{\rho}$. As a result, by Lemma 4.4, the judgement above implies $[\![\Gamma]\!]_{\rho} + [\![A]\!]_{\rho} \vdash J_i \triangleright P_i :: \mathcal{B}_i$.

Because this holds for any $i \in I$, repeated use of B-OR yields a derivation of $[\![\Gamma]\!]_{\rho} + [\![A]\!]_{\rho} \vdash D :: \mathcal{B}$. Lastly, because this holds for any $\mathcal{B} \in (\!|A|\!)_{\rho}$, B-GEN yields $[\![\Gamma]\!]_{\rho} + [\![A]\!]_{\rho} \vdash D :: [\![A]\!]_{\rho}$.

Applying the induction hypothesis to the fourth premise yields $[\![\Gamma]\!]_{\rho} + [\![A]\!]_{\rho} \vdash P$. Apply B-DEF to conclude.

Case WEAKEN. The second premise gives $\rho \vdash C'$. Thus, the induction hypothesis may be applied to the first premise, yielding the desired judgement.

Case $\exists$ INTRO. We have $\rho \vdash \exists\bar{\alpha}.C$. Then, there exists an assignment $\rho'$ such that $(\rho' \setminus \bar{\alpha} = \rho \setminus \bar{\alpha}) \wedge \rho' \vdash C$. Considering the second premise, we have $[\![\Gamma]\!]_{\rho'} = [\![\Gamma]\!]_{\rho}$ and $\rho'(\mathcal{J}) = \rho(\mathcal{J})$. Thus, applying the induction hypothesis to the first premise and to $\rho'$ yields the desired judgement.

This proof is, in our opinion, fairly readable. In fact, all cases except DEF are next to trivial.

In the DEF case, we must show that the definition $D$ has type $[\![A]\!]_{\rho}$, where $A = \forall\bar{\alpha}[C]\twoheadrightarrow B$. Because B($T$) has extensional polymorphism (i.e. rule B-GEN), it suffices to show that it has every type $\mathcal{B} \in \Pi[\![A]\!]_{\rho}$. Notice how we must "cut $\mathcal{B}$ into pieces" $\mathcal{B}_i$, corresponding to each clause $J_i$, in order to make use of the per-clause generalization criterion. We use the induction hypothesis at the level of each clause, then recombine the resulting type derivations using B-OR. Notice how we use Lemma 4.4; proving an environment strengthening lemma at the level of JOIN($X$) would be much more cumbersome.

The eight non-syntax-directed rules are easily proven correct. Indeed, their conclusion denotes fewer (SUB-NAME, SUB-DEF, WEAKEN) or exactly the same ($\exists$ INTRO) judgements in B($T$) as their premise. In a syntactic proof, the presence of these rules would require several normalization lemmas.

**Corollary 5.11.** *No well-typed process gets faulty through reduction.*

*Proof.* Assume $C, \Gamma \vdash P$. Because $C$ must be satisfiable, it must have at least one solution $\rho$. By Theorem 5.10, $[\![\Gamma]\!]_{\rho} \vdash P$ holds in B($T$). The result follows by Theorems 4.10 and 4.12.

## 6 Type Inference

### 6.1 Trouble with Generalization

Two severe problems quickly arise when attempting to define a *complete* type inference procedure for JOIN($X$). Both are caused by the *fragility* of the logical generalization criterion.

**Non-determinism.** To begin with, the criterion is non-deterministic. It states a sufficient condition for a given choice of $\bar{\alpha}$ to be correct. However, there seems to be, in general, no best choice. Consider the environment fragment $B = (a :$

$\langle\alpha\rangle; b : \langle\beta\rangle)$ under the constraint $\alpha \cap \beta = \varnothing$ (assuming the logic $X$ offers such a constraint, e.g. $X$ is a set constraint logic). The constraint creates a correlation between the names $a$ and $b$. Is it best to generalize $\alpha$, leaving $\beta$ monomorphic, or to do the converse?

**Non-monotonicity.** More subtly, *strengthening* the constraint $C$ may, in some cases, cause apparent correlations to *disappear*. Consider the environment fragment $B = (a : \alpha; b : \beta)$ under the constraint $\gamma?\alpha = \beta$ (assuming the logic $X$ offers such a constraint, to be read "if $\gamma$ is non-$\bot$, then $\alpha$ must equal $\beta$"). There is a correlation between $a$ and $b$, because, *in certain cases* (that is, when $\gamma \neq \bot$), $\alpha$ and $\beta$ must coincide. However, let us now add the constraint $\gamma = \bot$. We obtain $\gamma?\alpha = \beta \wedge \gamma = \bot$, which is logically equivalent to $\gamma = \bot$. It is clear that, under the new constraint, $a$ and $b$ are no longer correlated. So, the set of generalizable type variables may *increase* as the constraint $C$ is made more restrictive.

Given a definition $D$, a natural type inference algorithm will infer the weakest constraint $C$ under which it is well-typed, then will use $C$ to determine which type variables may be generalized. Because of non-monotonicity, the algorithm may find apparent correlations which would disappear if the constraint were deliberately strengthened. However, there is no way for the algorithm to guess if and how it should do so.

These remarks show that it is difficult to define a *complete* type inference algorithm, i.e. one which provably yields a single, most general typing.

Previous works [3, 6] use a similar type-based criterion, yet report no difficulty with type inference. This leads us to conjecture that these problems do not arise when subtyping is interpreted as equality and no custom constraint forms are available. This may be true for other constraint logics as well. Thus, a partial solution would be to define a type inference procedure only for those logics, taking advantage of their particular structure to prove its completeness.

In the general case, i.e. under an arbitrary choice of $X$, we know of no solution other than to abandon the logical criterion. We suggest replacing it with a much more naïve one, based on the structure of the *definition* itself, rather than on type information. One possible such criterion is given in Fig. 4. It simply consists in refusing generalization entirely if the definition involves *any* synchronization, i.e. if any join-pattern defines more than one name. (It is possible to do slightly better, e.g. by generalizing all names not involved in a synchronization between two messages of non-zero arity.) It is clearly safe with respect to the previous criterion.

The new criterion is deterministic, and impervious to changes in $C$, since it depends solely on the structure of the definition $D$. It is the analogue of the so-called *value restriction*, suggested by Wright [9], now in use in most ML implementations. Experience with ML suggests that such a restriction is tolerable in practice; a quick experiment shows that all of the sample code bundled with JoCaml [1] is well-typed under it.

In the following, we adopt the restricted Def rule of Fig. 4.

$$\frac{\begin{array}{c}\textsc{Def}\\ C, \Gamma + B \vdash (J_i \triangleright P_i)^{i\in I} :: B \qquad \bar{\alpha} \cap \mathrm{fv}(\Gamma) = \varnothing \\ (\exists i \in I \quad |\,\mathrm{dn}(J_i)\,| > 1) \Rightarrow \bar{\alpha} = \varnothing \\ C', \Gamma + \forall\bar{\alpha}[C]\twoheadrightarrow B \vdash P \qquad C' \Vdash C\end{array}}{C', \Gamma \vdash \mathtt{def}\ (J_i \triangleright P_i)^{i\in I}\ \mathtt{in}\ P}$$

**Fig. 4.** Definitive Def rule

## 6.2 A Type Inference Algorithm

Fig. 5 gives a set of syntax-directed type inference rules. Again, in every judgement $C, \Gamma \vdash_I \mathcal{J}$, it is understood that $C$ must be satisfiable. The rules implicitly describe an algorithm, whose inputs are an environment $\Gamma$ and a sub-term $u$, $D$ or $P$, and whose output, in case of success, is a judgement. Rule i-Or uses the following notation:

**Definition 6.1.** *The* least upper bound *of $B_1$ and $B_2$, written $B_1 \sqcup B_2$, is a pair of a monomorphic environment and a constraint. It is defined by:*

$$B_1 \sqcup B_2 = (u : \alpha_u)^{u\in U}, \qquad \bigwedge_{i\in\{1,2\}, u\in\mathrm{dom}(B_i)} B_i(u) \leq \alpha_u$$

*where $U = \mathrm{dom}(B_1) \cup \mathrm{dom}(B_2)$ and the type variables $(\alpha_u)^{u\in U}$ are fresh.*

Following [8], we have saturated every type inference judgement by existential quantification. Although slightly verbose, this style nicely shows which type variables are local to a sub-derivation, yielding the following invariant:

**Lemma 6.2.** *If $C, \Gamma \vdash_I \mathcal{J}$ holds, then $\mathrm{fv}(C) \subseteq \mathrm{fv}(\Gamma, \mathcal{J})$ and $\mathrm{fv}(\mathcal{J}) \cap \mathrm{fv}(\Gamma) = \varnothing$.*

We now prove the type inference rules correct and complete with respect to JOIN($X$).

**Theorem 6.3 (Soundness).** *$C, \Gamma \vdash_I \mathcal{J}$ implies $C, \Gamma \vdash \mathcal{J}$.*

*Proof.* By structural induction on the derivation of the input judgement. For the sake of conciseness, the induction hypothesis is applied silently.

Case i-Inst. Because the scope of the bound type variables $\bar{\alpha}$ is the same in the premise and in the conclusion, namely $C$ and $\alpha$ ($\beta$ is to be taken fresh, i.e. outside of the scope of $\bar{\alpha}$), we can perform $\alpha$-conversion on both judgements and require $\bar{\alpha} \cap \mathrm{fv}(\Gamma) = \varnothing$. By Inst, we have $C, \Gamma \vdash u : \alpha$. By Weaken and Sub-Name, we obtain $C \wedge \alpha \leq \beta, \Gamma \vdash u : \beta$. The result follows by $\exists$ Intro.

Case i-Empty. $C$ implies **true**; apply Weaken.

Case i-Join. Applying Weaken to the first premise yields $C \wedge \bigwedge_{i\in I} \beta_i = \langle\vec{\alpha_i}\rangle, \Gamma + (\vec{u_i} : \vec{\alpha_i})^{i\in I} \vdash P$. Then, rule Join applies. Lastly, because the type variables $(\bar{\alpha}_i)^{i\in I}$ are taken fresh, they do not appear in $\Gamma$; the result follows by $\exists$ Intro.

**Names**

I-INST

$$\frac{\Gamma(u) = \forall\bar{\alpha}[C].\alpha \qquad \beta \text{ fresh}}{\exists\bar{\alpha}.(C \wedge \alpha \leq \beta), \Gamma \vdash_I u : \beta}$$

**Definitions**

I-EMPTY

$$\mathbf{true}, \Gamma \vdash_I \epsilon :: \vec{0}$$

I-JOIN

$$\frac{C, \Gamma + (\vec{u_i} : \vec{\alpha_i})^{i\in I} \vdash_I P \qquad (\vec{\alpha}_i)^{i\in I}, (\beta_i)^{i\in I} \text{ fresh}}{\exists(\bar{\alpha}_i)^{i\in I}.(C \wedge \bigwedge_{i\in I} \beta_i = \langle\vec{\alpha_i}\rangle), \Gamma \vdash_I (x_i \langle \vec{u_i} \rangle)^{i\in I} \triangleright P :: (x_i : \beta_i)^{i\in I}}$$

I-OR

$$\frac{\begin{array}{cc} C_1, \Gamma \vdash_I D_1 : B_1 & C_2, \Gamma \vdash_I D_2 : B_2 \\ B, C = B_1 \sqcup B_2 & \bar{\beta} = \text{fv}(B_1, B_2) \end{array}}{\exists\bar{\beta}.(C_1 \wedge C_2 \wedge C), \Gamma \vdash_I D_1, D_2 :: B}$$

**Processes**

I-NULL

$$\mathbf{true}, \Gamma \vdash_I 0$$

I-PAR

$$\frac{C_1, \Gamma \vdash_I P \qquad C_2, \Gamma \vdash_I Q}{C_1 \wedge C_2, \Gamma \vdash_I P \mid Q}$$

I-MSG

$$\frac{C, \Gamma \vdash_I u : \beta \qquad \vec{C}, \Gamma \vdash_I \vec{v} : \vec{\alpha}}{\exists\beta\bar{\alpha}.(C \wedge \vec{C} \wedge \beta = \langle\vec{\alpha}\rangle), \Gamma \vdash_I u \langle \vec{v} \rangle}$$

I-DEF

$$\frac{\begin{array}{c} B \text{ fresh} \qquad \bar{\beta} = \text{fv}(B) \\ C_1, \Gamma + B \vdash_I (J_i \triangleright P_i)^{i\in I} :: B' \\ \bar{\beta}' = \text{fv}(B') \qquad C_2 = \exists\bar{\beta}'.(C_1 \wedge B' \leq B) \\ \text{if } \exists i \in I \quad |\,\text{dn}(J_i)\,| > 1 \text{ then } \bar{\alpha} = \varnothing \text{ else } \bar{\alpha} = \bar{\beta} \\ C_3, \Gamma + \forall\bar{\alpha}[C_2]\twoheadrightarrow B \vdash_I P \end{array}}{\exists\bar{\beta}.(C_2 \wedge C_3), \Gamma \vdash_I \texttt{def } (J_i \triangleright P_i)^{i\in I} \texttt{ in } P}$$

**Fig. 5.** Type inference

Case I-OR. By WEAKEN, $C_1 \wedge C_2 \wedge C, \Gamma \vdash D_i :: B_i$ holds for $i \in \{1,2\}$. Furthermore, by Definition 6.1, $B$ is of the form $B = B'_1 \oplus B'_2$, where $C_1 \wedge C_2 \wedge C \Vdash B_i \leq B'_i$ for $i \in \{1,2\}$. By SUB-DEF, $C_1 \wedge C_2 \wedge C, \Gamma \vdash D_i :: B'_i$ holds. OR then yields $C_1 \wedge C_2 \wedge C, \Gamma \vdash D_1, D_2 :: B$. By Lemma 6.2, $\bar{\beta}$ does not appear free in $\Gamma$; neither does it appear in $\text{fv}(B)$, since $B$ is made up of fresh variables. Apply $\exists$ INTRO to conclude.

Case I-NULL. $C$ implies **true**; apply WEAKEN.

Case I-PAR. Apply WEAKEN to each premise, then PAR.

Case I-MSG. Let $C' = C \wedge \vec{C} \wedge \beta = \langle\vec{\alpha}\rangle$. By WEAKEN, we have $C', \Gamma \vdash u : \beta$ and $C', \Gamma \vdash \vec{v} : \vec{\alpha}$. Then, MSG applies, yielding $C', \Gamma \vdash u \langle \vec{v} \rangle$. By Lemma 6.2, $\beta$ and $\bar{\alpha}$ do not appear in $\Gamma$; the result follows by $\exists$ INTRO.

Case I-DEF. WEAKEN and SUB-DEF may be applied to the first premise, yielding $C_1 \wedge B' \leq B, \Gamma + B \vdash D : B$. By Lemma 6.2, $\bar{\beta}'$ does not appear free in $\Gamma$ or $B$. Thus, $\exists$ INTRO yields $C_2, \Gamma + B \vdash D : B$. The sixth premise, together with the freshness of $\bar{\beta}$, implies the second and third premises of rule Def (Fig. 4). Applying Weaken to the last premise, we have $C_2 \wedge C_3, \Gamma + \forall\bar{\alpha}[C_2]{\twoheadrightarrow}B \vdash P$. Thus, Def yields $C_2 \wedge C_3, \Gamma \vdash \texttt{def}\ D\ \texttt{in}\ P$. Apply $\exists$ Intro to conclude.

To allow an inductive proof, completeness must be given a quite general statement; the following formulation was suggested to us by Martin Sulzmann. For conciseness, we use a pseudo-constraint $\mathcal{J} \leq \mathcal{J}'$, defined as follows: $u : \alpha \leq u : \alpha'$ stands for $\alpha \leq \alpha'$; $D : B \leq D : B'$ stands for $B \leq B'$; $P \leq P$ stands for **true**.

**Theorem 6.4 (Completeness).** *Assume $C, \Gamma \vdash \mathcal{J}$ and $C'' \Vdash \Gamma' \leq \Gamma$ and $C'' \Vdash C$. Then, there exist $C'$ and $\mathcal{J}'$ such that $C', \Gamma' \vdash_I \mathcal{J}'$ and $C'' \Vdash \exists \mathrm{fv}(\mathcal{J}').(C' \wedge \mathcal{J}' \leq \mathcal{J})$.*

*Proof.* By induction on the derivation of the input judgement. We recall that, by definition, $C \Vdash \forall\bar{\alpha}_1[C_1].\alpha_1 \leq \forall\bar{\alpha}_2[C_2].\alpha_2$ holds if and only if $\rho \vdash C$ implies $[\![\forall\bar{\alpha}_1[C_1].\alpha_1]\!]_\rho \supseteq [\![\forall\bar{\alpha}_2[C_2].\alpha_2]\!]_\rho$, that is, if and only if

$$\exists\bar{\alpha}_2.C \wedge C_2 \Vdash \exists\bar{\alpha}_1.(C_1 \wedge \alpha_1 \leq \alpha_2)$$

assuming $\bar{\alpha}_i$ does not appear free in $\bar{\alpha}_j$, $C_j$ or $\alpha_j$ when $i \neq j$. This is extended point-wise to environments.

Case Inst. We assume $\Gamma(u) = \forall\bar{\alpha}[C].\alpha$ and $\Gamma'(u) = \forall\bar{\alpha}'[C'].\alpha'$. Without loss of generality, we assume $\bar{\alpha}$ and $\bar{\alpha}'$ are taken disjoint. The original judgement, derived by Inst, is $C, \Gamma \vdash u : \alpha$, while, by i-Inst, we may derive $\exists\bar{\alpha}'.(C' \wedge \alpha' \leq \beta), \Gamma' \vdash u : \beta$, where $\beta$ is a fresh variable. Thus, there remains to prove that

$$C'' \Vdash \exists\beta.(\exists\bar{\alpha}'.(C' \wedge \alpha' \leq \beta) \wedge \beta \leq \alpha)$$

Now, because $C'' \Vdash \Gamma' \leq \Gamma$, we have $C'' \Vdash \forall\bar{\alpha}'[C'].\alpha' \leq \forall\bar{\alpha}[C].\alpha$, which, by definition of $\Vdash$, is $\exists\bar{\alpha}.C'' \wedge C \Vdash \exists\bar{\alpha}'.(C' \wedge \alpha' \leq \alpha)$. Because $C'' \Vdash C$, this assertion may be weakened to $C'' \Vdash \exists\bar{\alpha}'.(C' \wedge \alpha' \leq \alpha)$. The result follows.

Case Sub-Name. Applying the induction hypothesis to the first premise yields $C'$ and $\beta'$ such that $C', \Gamma' \vdash_I u : \beta'$ and $C'' \Vdash \exists\beta'.(C' \wedge \beta' \leq \alpha')$. Without loss of generality, we assume $\beta'$ is distinct from $\alpha$ and $\alpha'$. The second premise is $C \Vdash \alpha' \leq \alpha$. Given $C'' \Vdash C$, there follows $C'' \Vdash \exists\beta'.(C' \wedge \beta' \leq \alpha)$.

Case Empty. Immediate.

Case Join. Pick fresh variables $\vec{\alpha}'_i, \beta'_i$ for $i \in I$. Define $C_1 = C'' \wedge \bigwedge_{i\in I}(\vec{\alpha}'_i \leq \vec{\alpha}_i)$. Then, $C_1 \Vdash \Gamma' + (u_i : \vec{\alpha}'_i)^{i\in I} \leq \Gamma + (u_i : \vec{\alpha}_i)^{i\in I}$ holds. The first premise of Join is $C, \Gamma + (u_i : \vec{\alpha}_i)^{i\in I} \vdash P$. Applying the induction hypothesis to it yields $C'$ such that $C', \Gamma' + (u_i : \vec{\alpha}'_i)^{i\in I} \vdash_I P$ and $C_1 \Vdash C'$. Then, i-Join yields

$$\exists(\bar{\alpha}'_i)^{i\in I}.(C' \wedge \bigwedge_{i\in I} \beta'_i = \langle\vec{\alpha}'_i\rangle), \Gamma' \vdash_I J \triangleright P :: (x_i : \beta'_i)^{i\in I}$$

Thus, there remains to prove

$$C'' \Vdash \exists\bar{\beta}'.\left(\exists(\bar{\alpha}'_i)^{i\in I}.(C' \wedge \bigwedge_{i\in I} \beta'_i = \langle\vec{\alpha}'_i\rangle) \wedge \bigwedge_{i\in I}(\beta'_i \leq \beta_i)\right)$$

Because $C_1 \Vdash C'$, and considering our freshness hypotheses, it suffices to check that

$$C'' \Vdash \exists\bar{\beta}'(\bar{\alpha}'_i)^{i\in I}.(C'' \wedge \bigwedge_{i\in I} \vec{\alpha}'_i \leq \vec{\alpha}_i \wedge \beta'_i = \langle\vec{\alpha}'_i\rangle \wedge \beta'_i \leq \beta_i)$$

which follows from JOIN's second premise, namely $\forall i \in I \quad C \Vdash \beta_i = \langle\vec{\alpha}_i\rangle$, and from $C'' \Vdash C$.

Case OR. The premises are $C, \Gamma \vdash D_i :: B_i$ for $i \in \{1,2\}$. Applying the induction hypothesis yields $C'_i, B'_i$ such that $C'_i, \Gamma' \vdash_I D_i : B'_i$ and $C'' \Vdash \exists\bar{\beta}'_i.(C'_i \wedge B'_i \leq B_i)$, where $\bar{\beta}'_i = \mathrm{fv}(B'_i)$. Without loss of generality, we may assume that $\bar{\beta}'_i$ does not appear in $C'_j$, $B'_j$ and $B_j$ when $i \neq j$, so that

$$C'' \Vdash \exists\bar{\beta}'_1\bar{\beta}'_2.(C'_1 \wedge C'_2 \wedge B'_1 \leq B_1 \wedge B'_2 \leq B_2)$$

holds. Define $B', C'$ as $B'_1 \sqcup B'_2$. Then, I-OR yields

$$\exists\bar{\beta}'_1\bar{\beta}'_2.(C'_1 \wedge C'_2 \wedge C'), \Gamma' \vdash_I D_1, D_2 :: B'$$

Thus, there remains to prove

$$C'' \Vdash \exists\bar{\beta}'.(\exists\bar{\beta}'_1\bar{\beta}'_2.(C'_1 \wedge C'_2 \wedge C') \wedge B' \leq B_1 \oplus B_2)$$

which, considering our freshness hypotheses and the definition of $B'_1 \sqcup B'_2$, is easily seen to hold.

Case SUB-DEF. Similar to SUB-NAME.

Cases NULL, PAR. Immediate.

Case MSG. Applying the induction hypothesis to the first premise yields $C'$ and $\beta'$ such that $C', \Gamma' \vdash_I u : \beta'$ and $C'' \Vdash \exists\beta'.(C' \wedge \beta' \leq \beta)$. The judgement $C', \Gamma' \vdash_I u : \beta'$ must have been produced by I-INST; a look at this rule shows that $\beta'$ appears only in a single constraint of $C'$, where it is given a lower bound. As a result, $C'' \Vdash \exists\beta'.(C' \wedge \beta' \leq \beta)$ implies $C'' \Vdash \exists\beta'.(C' \wedge \beta' = \beta)$. Similarly, assuming $\vec{v} = (v_i)^{i\in I}$ and $\vec{\alpha} = (\alpha_i)^{i\in I}$, applying the induction hypothesis to the $i$-th component ($i \in I$) of the second premise yields $C'_i$ and $\alpha'_i$ such that $C'_i, \Gamma' \vdash_I v_i : \alpha'_i$ and $C'' \Vdash \exists\alpha'_i.(C'_i \wedge \alpha'_i = \alpha_i)$.

As in Case OR above, our freshness hypotheses allow these entailment assertions to be combined, yielding

$$C'' \Vdash \exists\beta'\bar{\alpha}'.(C' \wedge \beta' = \beta \wedge \bigwedge_{i\in I}(C'_i \wedge \alpha'_i = \alpha_i))$$

where $\bar{\alpha}'$ is $(\alpha'_i)^{i\in I}$. To combine the type inference judgements, we apply rule I-MSG, yielding

$$\exists\beta'\bar{\alpha}'.(C' \wedge \bigwedge_{i\in I} C'_i \wedge \beta' = \langle\vec{\alpha}'\rangle), \Gamma' \vdash_I u\,\langle\,\vec{v}\,\rangle$$

Thus, there remains to prove that

$$C'' \Vdash \exists\beta'\bar{\alpha}'.(C' \wedge \bigwedge_{i\in I} C'_i \wedge \beta' = \langle\vec{\alpha}'\rangle)$$

which follows from the entailment assertion above, from the third premise $C \Vdash \beta = \langle \vec{\alpha} \rangle$ and from the hypothesis $C'' \Vdash C$.

Case Def. Our hypotheses are as in Fig. 4, except we take the last premise to be $C' \Vdash \exists \bar{\alpha}.C$. As mentioned in Sect. 5.1, this does not affect the type system's expressive power, but allows us, without loss of generality, to assume that $\bar{\alpha}$ does not appear free in $\Gamma$, $\Gamma'$, $C'$, $C''$.

We assume $C'' \Vdash C'$. Pick a fresh $B'$; define $\bar{\beta}' = \mathrm{fv}(B')$. Define $C_0 = C'' \wedge B' = B$. We have $C_0 \Vdash \Gamma' + B' \leq \Gamma + B$. Furthermore, because $C' \Vdash C$, $C_0 \Vdash C$ holds. Thus, we may apply the induction hypothesis to the first premise, yielding

$$C_1, \Gamma' + B' \vdash_I D :: B''$$

where $C_0 \Vdash \exists \bar{\beta}''.(C_1 \wedge B'' \leq B)$ and $\bar{\beta}'' = \mathrm{fv}(B'')$. We assume, without loss of generality, that $\bar{\beta}''$ is fresh with respect to $B$ and $B'$.

Define $C_2 = \exists \bar{\beta}''.(C_1 \wedge B'' \leq B')$. Because $B$ and $B'$ are interchangeable under $C_0$, and considering the above freshness hypothesis, $C_0 \Vdash C_2$ holds.

Because $\bar{\beta}'$ appears neither in $C''$ nor in $B$, we have $C'' \Vdash \exists \bar{\beta}'.(B' = B \wedge C_0)$. Recalling $C_0 \Vdash \exists \bar{\beta}''.(C_1 \wedge B'' \leq B)$, this implies

$$C'' \Vdash \exists \bar{\beta}'.(B' = B \wedge \exists \bar{\beta}''.(C_1 \wedge B'' \leq B))$$

which, given the freshness of $\bar{\beta}''$, can be written

$$C'' \Vdash \exists \bar{\beta}'\bar{\beta}''.(C_1 \wedge B'' \leq B' = B)$$

Define $\bar{\alpha}'$ as $\varnothing$ if $\exists i \in I \quad |\,\mathrm{dn}(J_i)\,| > 1$, and as $\bar{\beta}'$ otherwise. We claim that

$$C_0 \Vdash \Gamma' + \forall \bar{\alpha}'[C_2] \twoheadrightarrow B' \leq \Gamma + \forall \bar{\alpha}[C] \twoheadrightarrow B$$

If $\bar{\alpha} = \bar{\alpha}' = \varnothing$, this is an immediate consequence of $C_0 \Vdash B' = B$ and $C_0 \Vdash C_2$. Otherwise, it suffices to check, for every $(u : \beta') \in B'$ and $(u : \beta) \in B$, that

$$C_0 \Vdash \forall \bar{\alpha}'[C_2].\beta' \leq \forall \bar{\alpha}[C].\beta$$

which, by definition of $\Vdash$, is equivalent to

$$\exists \bar{\alpha}.C_0 \wedge C \Vdash \exists \bar{\alpha}'.(C_2 \wedge \beta' \leq \beta)$$

Recalling that $\bar{\alpha}' = \bar{\beta}'$, and lifting $C_2$'s outermost existential quantifier to the toplevel, this can be written

$$\exists \bar{\alpha}.C_0 \wedge C \Vdash \exists \bar{\beta}'\bar{\beta}''.(C_1 \wedge B'' \leq B' \wedge \beta' \leq \beta)$$

By weakening the left-hand side to $\exists \bar{\alpha}.C''$, then recalling that $\bar{\alpha}$ does not appear free in $C''$; by recalling that $\beta'$ and $\beta$ are corresponding elements of $B'$ and $B$, this may strengthened to

$$C'' \Vdash \exists \bar{\beta}'\bar{\beta}''.(C_1 \wedge B'' \leq B' \leq B)$$

which we have already proven above. Thus, our claim holds.

This allows us to apply the induction hypothesis to the penultimate premise of DEF, yielding $C_3$ such that $C_3, \Gamma' + \forall\bar{\alpha}'[C_2]\twoheadrightarrow B' \vdash_I P$ and $C' \Vdash C_3$. We may then apply I-DEF, yielding $\exists\bar{\beta}'.(C_2 \wedge C_3), \Gamma' \vdash \mathtt{def}\ D\ \mathtt{in}\ P$. Thus, there remains to prove

$$C'' \Vdash \exists\bar{\beta}'.(C_2 \wedge C_3)$$

This can be shown by recalling $C'' \Vdash \exists\bar{\beta}'.(B' = B \wedge C_0)$. Because $C_0 \Vdash C_2$ and $C_0 \Vdash C'' \Vdash C' \Vdash C_3$, this implies $C'' \Vdash \exists\bar{\beta}'.(B' = B \wedge C_2 \wedge C_3)$, whence the result.

Case WEAKEN. The second premise is $C \Vdash C'$. Because $C'' \Vdash C$, we have $C'' \Vdash C'$. Thus, the induction hypothesis can be applied to the first premise, yielding the desired result.

Case $\exists$ INTRO. Assume $C'' \Vdash \exists\bar{\alpha}.C$. Without loss of generality, we assume that $\bar{\alpha}$ does not appear free in $C''$ or $\Gamma'$. Applying the induction hypothesis to $C'' \wedge C$ and to the first premise, we obtain $C'$ and $\mathcal{J}'$ such that $C', \Gamma' \vdash_I \mathcal{J}'$ and

$$C'' \wedge C \Vdash \exists\bar{\beta}'.(C' \wedge \mathcal{J}' \leq \mathcal{J})$$

where $\bar{\beta}' = \mathrm{fv}(\mathcal{J}')$. Introducing an additional existential quantifier on both sides, we obtain

$$\exists\bar{\alpha}.(C'' \wedge C) \Vdash \exists\bar{\alpha}.\exists\bar{\beta}'.(C' \wedge \mathcal{J}' \leq \mathcal{J})$$

Because $\bar{\alpha}$ does not occur in $C''$, and because $C'' \Vdash \exists\bar{\alpha}.C$, the left-hand side is equivalent to $C''$. Furthermore, by Lemma 6.2, the free variables of $C'$ and $\mathcal{J}'$ must occur free in $\Gamma'$, hence cannot appear in $\bar{\alpha}$. Thus, we have

$$C'' \Vdash \exists\bar{\beta}'.(C' \wedge \mathcal{J}' \leq \mathcal{J})$$

which is the desired result.

Soundness and completeness of the type inference rules may now be re-stated in a more concise way. For the sake of simplicity, we limit the statement to the case of processes (omitting that of names and definitions).

**Theorem 6.5.** *$C, \Gamma \vdash_I P$ implies $C, \Gamma \vdash P$. Conversely, if $C, \Gamma \vdash P$ holds, then there exists a constraint $C'$ such that $C', \Gamma \vdash_I P$ and $C \Vdash C'$.*

*Proof.* By specializing Theorems 6.3 and 6.4.

## 7 Discussion

JOIN($X$) is closely related to HM($X$) [5, 7], a similar type system aimed at purely functional languages. It also draws inspiration from previous type systems for the join-calculus [3, 6], which were purely unification-based. JOIN($X$) is an attempt to bring together these two orthogonal lines of research.

Our results are partly negative: under a natural generalization criterion, the existence of principal typings is problematic. This leads us, in the general case, to suggest a more drastic restriction. Nevertheless, the logical criterion may still be useful under certain specific constraint logics, where principal typings can still be achieved, or in situations where their existence is not essential (e.g. in program analysis).

We have not discussed extending the calculus with primitive operations and associated $\delta$-rules. Such an extension is straightforward. It requires assigning a set of monotypes to each operation $p$ in B($T$), and extending the subject reduction and progress proofs; then, assigning a type scheme to $p$ in JOIN($X$), and showing that all elements of its denotation are valid monotypes for $p$ in B($T$).

To establish type safety, we interpret typing judgements as (sets of) judgements in an underlying system, which is given a syntactic soundness proof. The former step, by giving a logical view of polymorphism and constraints, aptly expresses our intuitions about these notions, yielding a concise proof. The latter is a matter of routine, because the low-level type system is simple. Thus, both logic and syntax are put to best use. We have baptized this approach *semi-syntactic*; we feel it is perhaps not publicized enough.

One may argue that this approach only yields a type safety result, whereas subject reduction and progress [10] are more precise properties. This is true, but let us ask: is it worth it? The semi-syntactic approach may afford a more modular proof, where subtle and interesting aspects (e.g. constraints, generalization) are clearly separated from administrative ones (e.g. substitution lemmas, etc.). For the record, we have established a subject reduction property for JOIN($X$), using a restricted operational semantics along the lines of [6]. The proof does not have such pleasant modular structure, and has shown rather fragile when confronted to changes in the typing rules, whereas the semi-syntactic proof turns out to be much easier to incrementally update.

## Acknowledgements

Alexandre Frey suggested the use of extensional polymorphism in the intermediate type system B($T$). Martin Sulzmann kindly provided a proof of completeness of constraint-based type inference in HM($X$), which was helpful in our own completeness proof. We would also like to acknowledge Martin Odersky and Didier Rémy for stimulating discussions.

## References

[1] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99)*, pages 22–29, Palm Springs, California, October 1999. URL: `http://para.inria.fr/~conchon/publis/asa99.ps.gz`.

[2] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, 1996. URL: `http://pauillac.inria.fr/~fournet/papers/popl-96.ps.gz`.

[3] Cédric Fournet, Luc Maranget, Cosimo Laneve, and Didier Rémy. Implicit typing à la ML for the join-calculus. In *8th International Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *Lecture Notes in Computer Science*, pages 196–212, Warsaw, Poland, 1997. Springer. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/typing-join.ps.gz`.

[4] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[5] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. URL: `http://www.cs.mu.oz.au/~sulzmann/publications/tapos.ps`.

[6] Martin Odersky, Christoph Zenger, Matthias Zenger, and Gang Chen. A functional view of join. Technical Report ACRC-99-016, University of South Australia, 1999. URL: `http://lampwww.epfl.ch/~czenger/papers/tr-acrc-99-016.ps.gz`.

[7] Martin Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000. URL: `http://www.cs.mu.oz.au/~sulzmann/publications/diss.ps.gz`.

[8] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999. URL: `http://www.ps.uni-sb.de/~mmueller/papers/hm-constraints.ps.gz`.

[9] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

[10] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994. URL: `http://www.cs.rice.edu/CS/PLT/Publications/ic94-wf.ps.gz`.