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

Sylvain Conchon, François Pottier

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.

1 Introduction

The join-calculus [3] is a name-passing process calculus related to the asynchronous π-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 [4], `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.* [4] 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.* [6], 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 α and β. 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* [10].

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,

$$P \mid Q \rightleftharpoons Q \mid P \qquad D_1, D_2 \rightleftharpoons D_2, D_1$$
$$P \mid 0 \rightleftharpoons P \qquad D, \epsilon \rightleftharpoons D$$
$$P \mid (Q \mid R) \rightleftharpoons (P \mid Q) \mid R \qquad D_1, (D_2, D_3) \rightleftharpoons (D_1, D_2), D_3$$

$$(\texttt{def}\ D\ \texttt{in}\ P) \mid Q \rightleftharpoons \texttt{def}\ D\ \texttt{in}\ (P \mid Q) \qquad \text{if } \mathrm{dn}(D) \cap \mathrm{fn}(Q) = \varnothing$$
$$\texttt{def}\ D_1\ \texttt{in}\ \texttt{def}\ D_2\ \texttt{in}\ P \rightleftharpoons \texttt{def}\ D_1, D_2\ \texttt{in}\ P \qquad \text{if } \mathrm{fn}(D_1) \cap \mathrm{dn}(D_2) = \varnothing$$
$$\texttt{def}\ D, J \triangleright P\ \texttt{in}\ Q \mid \varphi J \to \texttt{def}\ D, J \triangleright P\ \texttt{in}\ Q \mid \varphi P \qquad \text{if } \mathrm{dom}(\varphi) = \mathrm{ln}(J)$$

Fig. 1. Operational semantics

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.

By lack of space, we omit all proofs, except that of the main type soundness theorem (Theorem 5.9). The interested reader is referred to [2].

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.

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

The *defined names* $\mathrm{dn}(J)$ (resp. $\mathrm{dn}(D)$) of a join-pattern J (resp. of a definition D) are the channels defined by it. In a process $\texttt{def}\ D\ \texttt{in}\ P$, the defined names of D are bound within D and P. More details are given in [2].

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

3 Notation

Definition 3.1. *Given a set T, a* T-environment*, usually denoted Γ, is a partial mapping from $\mathcal{N}$ into T. If $N \subseteq \mathcal{N}$, $\Gamma|_N$ denotes the restriction of Γ 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 Γ and Γ' 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 [5, 6].

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

Names

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

Definitions

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

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

Processes

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

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

Fig. 2. The system B(T)

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 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.

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.

The following lemmas will be used in the proof of Theorem 5.9. The first one allows weakening type judgements by strengthening their environment. The second one is technical.

Lemma 4.4. *If $\Gamma \vdash P$ and $\Gamma' \leq \Gamma$, then $\Gamma' \vdash P$.*

Lemma 4.5. *Assume $\Gamma \vdash (D, J \triangleright P) :: \mathcal{B}'$ and $\mathcal{B}'|_{\mathrm{dn}(J)} \leq \mathcal{B}$. Then $\Gamma \vdash J \triangleright P :: \mathcal{B}$.*

We establish type soundness for B(T) following the syntactic approach of Wright and Felleisen [11], i.e. by proving that B(T) enjoys *subject reduction* and *progress* properties.

Theorem 4.6 (Subject reduction). *$\Gamma \vdash P$ and $P \to P'$ imply $\Gamma \vdash P'$.*

Definition 4.7. *A process of the form* $\texttt{def}\ D, J \triangleright P\ \texttt{in}\ Q \mid u\langle \vec{v} \rangle$ *is* faulty *if J defines a message $u\langle \vec{y} \rangle$ where $\vec{v}$ and $\vec{y}$ have different arities.*

Theorem 4.8 (Progress). *No well-typed process is faulty.*

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) [6, 9, 8]. 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 5.9). 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 ρ from $\mathcal{V}$ into T, and whose second argument is a constraint C. The interpretation must be standard, i.e. satisfy the following laws:

$$\rho \vdash \mathbf{true}$$

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

($\rho \setminus \bar{\alpha}$ denotes the restriction of ρ 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 ρ of C satisfies C' as well.

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 α; we write $\sigma = \forall\bar{\alpha}[C].\alpha$. The type variables in $\bar{\alpha}$ are bound in σ; type schemes are considered equal modulo α-conversion. By abuse of notation, a type variable α 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 Γ 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 α-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 [6]. 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.*

Names

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

Definitions

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

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

Processes

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

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

Common

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

Fig. 3. The system JOIN(X) (with a tentative DEF rule)

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 Γ, 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 Γ.

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 [4, 7] 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 ρ, 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 σ 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 ρ, which gives meaning to the free type variables of σ.

Definition 5.6. *The denotation of an environment fragment $A = (u_i : \sigma_i)^{i\in I}$ under an assignment ρ, 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 ρ, 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 σ_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 ρ such that $\rho \vdash C$, $(\!|\forall\bar{\alpha}[C].B|\!)_\rho \supseteq (\!|\forall\bar{\alpha}[C]{\twoheadrightarrow}B|\!)_\rho$ holds.*

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 [4]. 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.* [7].

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.

Definition 5.8. *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 Γ is an $\mathcal{S}$-environment, then $[\![\Gamma]\!]_\rho$ is an S-environment.*

Theorem 5.9 (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 ρ 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 ρ' 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 ρ', yields $[\![\Gamma + B]\!]_{\rho'} \vdash D :: \rho'(B)$. By Lemma 4.5, 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 ρ', 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 ρ' 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 ρ' 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 5.10. *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 ρ. By Theorem 5.9, $[\![\Gamma]\!]_\rho \vdash P$ holds in B(T). The result follows by Theorems 4.6 and 4.8.

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.

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

Fig. 4. Definitive Def rule

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.

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 γ is non-$\bot$, then α must equal β"). There is a correlation between a and b, because, *in certain cases* (that is, when $\gamma \neq \bot$), α and β 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 [4, 7] 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 [10], 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.

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 Γ 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:*

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

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

Following [9], 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). For the sake of simplicity, we limit the statement to the case of processes (omitting that of names and definitions).

Theorem 6.3. *$C, \Gamma \vdash_I P$ 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 $C \Vdash C'$.*

7 Discussion

JOIN(X) is closely related to HM(X) [6, 8], a similar type system aimed at purely functional languages. It also draws inspiration from previous type systems for the join-calculus [4, 7], 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

Names

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

Definitions

$$\text{I-Empty}\quad \mathbf{true}, \Gamma \vdash_I \epsilon :: \vec{0}$$

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

$$\text{I-Or}\quad \frac{\begin{array}{c} C_1, \Gamma \vdash_I D_1 : B_1 \qquad C_2, \Gamma \vdash_I D_2 : B_2 \\ B, C = B_1 \sqcup B_2 \qquad \bar{\beta} = \text{fv}(B_1, B_2) \end{array}}{\exists \bar{\beta}.(C_1 \wedge C_2 \wedge C), \Gamma \vdash_I D_1, D_2 :: B}$$

Processes

$$\text{I-Null}\quad \mathbf{true}, \Gamma \vdash_I 0$$

$$\text{I-Par}\quad \frac{C_1, \Gamma \vdash_I P \qquad C_2, \Gamma \vdash_I Q}{C_1 \wedge C_2, \Gamma \vdash_I P \mid Q}$$

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

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

Fig. 5. Type inference

still be achieved, or in situations where their existence is not essential (e.g. in program analysis).

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.

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] Sylvain Conchon and François Pottier. JOIN(X): Constraint-based type inference for the join-calculus. Long version. URL: `http://pauillac.inria.fr/~fpottier/publis/conchon-fpottier-esop01-long.ps.gz`, April 2001.

[3] 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`.

[4] 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`.

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

[6] 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`.

[7] 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`.

[8] 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`.

[9] 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`.

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

[11] 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`.