Implicit typing à la ML for the join-calculus [*]

Cédric Fournet [†] Cosimo Laneve [‡]
Luc Maranget[†] Didier Rémy[†]

Abstract

We adapt the Damas-Milner typing discipline to the join-calculus. The main result is a new generalization criterion that extends the polymorphism of ML to join-definitions. We prove the correctness of our typing rules with regard to a chemical semantics. We also relate typed extensions of the core join-calculus to functional languages.

1 Introduction

The distributed implementation of concurrent calculi with message passing raises the problem of implementing communication channels, which finally reduces to the specification of channel managers. In order to reflect this need in the language itself, a new formalism has been recently introduced : the *join-calculus* [2]. This calculus is similar to Milner's asynchronous π-calculus, except that the operations of restriction, reception and replication are all combined into a single receptor definition. Such a combination yields better control over communication. In [2, 3], we relied on this locality property to model realistic distributed systems. In this paper, we propose a type system for the join-calculus whose simplicity owes much to locality.

The join-calculus is quite expressive (a lot of examples may be found in [2, 3]) and has been turned into a programming language. A convenient syntax has been provided for sequential composition, process migration and failure detection. A distributed implementation is under way and would benefit from static analyses of programs. A good static semantics should of course rely on a type system. The types we need should be expressive enough for most useful programs and easy to understand for programmers.

This goal is achieved by adapting the Damas-Milner typing discipline developed for ML [1] to the join calculus. From the typing point of view, definitions in the join-calculus are a generalized form of let expressions in ML and polymorphism can be introduced right after typechecking the clauses of a join-definition.

[*] This work is partly supported by the ESPRIT CONFER-2 WG-21836

[†] INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex France.

[‡] Dipartimento di Scienze dell'Informazione, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy

However, synchronization on channels is more demanding than plain function calls, as it interacts with polymorphism. Our main result is a generalization criterion for the join-calculus that addresses this issue. We prove the correctness of the resulting typing rules with regard to our concurrent semantics by adapting standard techniques to the chemical framework. Thus, without any change, the join calculus becomes a typed process calculus with implicit parametric polymorphism.

1.1 Polymorphism in the join-calculus

The join-calculus is essentially a name-passing calculus : port names are defined, then used as addresses in messages that convey other names. These messages are polyadic ; the type of a name carrying n objects of type $\tau_1, \ldots, \tau_n$ is written $\langle \tau_1, \ldots, \tau_n \rangle$. Traditional languages come with system-supplied primitives, which can be used in the programming practice. Similarly, we could assume system-supplied primitive names for a language based on the join-calculus, such as `print_int` that outputs its integer argument on the console. Then,

$$\texttt{def print\_two\_ints}\langle\texttt{x,y}\rangle \;\triangleright\; \texttt{print\_int}\langle\texttt{x}\rangle \mid \texttt{print\_int}\langle\texttt{y}\rangle$$

defines a new name `print_two_ints` that prints two integers ; more precisely, when the name `print_two_ints` receives a couple of arguments $\langle\texttt{x,y}\rangle$, it activates two processes $\texttt{print\_int}\langle\texttt{x}\rangle$ and $\texttt{print\_int}\langle\texttt{y}\rangle$ running concurrently. The type of the primitive `print_int` is $\langle\texttt{int}\rangle$ (i.e., a name that carries one integer) and the type of the new name `print_two_ints` is $\langle\texttt{int},\texttt{int}\rangle$ (i.e., a name that carries two integers).

In this context, a name with a polymorphic type in the join-calculus is reminiscent of a polymorphic function in ML : both don't need to perform fully type-specific operations on their arguments. Thus, the types of the arguments are not completely specified and unspecified parts are represented by type variables that stand for just any type. This framework is known as *parametric* polymorphism. For instance, consider the following definition :

$$\texttt{def apply}\langle\texttt{k,x}\rangle \;\triangleright\; \texttt{k}\langle\texttt{x}\rangle$$

The name `apply` takes two arguments `k` and `x` and activates the process $\texttt{k}\langle\texttt{x}\rangle$. Thus, if `x` is of type τ, then `k` must carry names of type τ, i.e. be of type $\langle\tau\rangle$. The name `apply` can be given type $\langle\langle\tau\rangle, \tau\rangle$ for any type τ. As in ML, this is emphasized by giving `apply` the type scheme $\forall \alpha. \langle\langle\alpha\rangle, \alpha\rangle$. Therefore `apply` can take arguments as `print_int` and `4`, by the call $\texttt{apply}\langle\texttt{print\_int,4}\rangle$, thereby instantiating α with the type `int`. Given another primitive `print_string`, another legitimate invocation $\texttt{apply}\langle\texttt{print\_string,'foo'}\rangle$ would instantiate α with the type `string`.

The join-calculus improves on ML by providing synchronization between join patterns (several messages in parallel). Consider, for instance, a variant of `apply` that receives `k` and `x` from different sources.

$$\texttt{def port}\langle\texttt{k}\rangle \mid \texttt{arg}\langle\texttt{x}\rangle \;\triangleright\; \texttt{k}\langle\texttt{x}\rangle$$

The concurrent activation of the co-defined names port and arg fires $\mathtt{k}\langle\mathtt{x}\rangle$. The names port and arg can be typed with $\langle\langle\alpha\rangle\rangle$ and $\langle\alpha\rangle$, respectively. Observe, however, that names port and arg are correlated, which is reflected by the use of the same type variable α in their types. This forbids to give port and arg the type schemes $\forall\alpha.\langle\alpha\rangle$ and $\forall\alpha.\langle\langle\alpha\rangle\rangle$. Otherwise, their types schemes could be instantiated independently, loosing their correlation. Clearly, sending the primitive print_string on port and an integer on arg would result in a run-time type error : attempting to print an integer as a string.

As a consequence, our generalization rule copes with synchronization in an abstract way : a type variable cannot be generalized if it appears free in the type of *several* co-defined names.

1.2 Overview

In section 2, we recall the syntax and semantics of the join-calculus and we present the type system. The original RCHAM used in [3] has a defect as regards typing ; we introduce a variant and we relate it to the original. In section 3, we establish the main result : we prove subject reduction in a chemical setting and we show that well-typed programs cannot go wrong at run-time. In section 4, we briefly discuss type inference, as implemented in our prototype compiler. In section 5, we extend the join-calculus with support for functions and expressions, an useful step towards an effective programming language. We generalize the type system accordingly. This extension provides a good basis for a detailed comparison with type systems for functional languages. In section 6, we compare our work with other type systems that have been proposed for concurrent calculi.

2 The typed join-calculus

2.1 Syntax

While names already provide enough expressiveness [2], it is convenient here to supplement names with constants that represent basic values 1, 2... ,'foo'... and primitives add, string_of_int, print, along with their basic types such as int, string.

For names, we assume given a set of port names $x \in \mathcal{N}$ and a set of constants $k \in \mathcal{K}$. We use $u \in \mathcal{N} \cup \mathcal{K}$ to denote a name in general. For types, we assume given a set of basic types $b \in \mathcal{T}$ and a set of type variables α.

$$\begin{array}{rl@{\qquad\qquad}rl}
P ::= & u\langle u_i{}^{i\in 1..p}\rangle & & \\
 & \mid \mathtt{def}\ D\ \mathtt{in}\ P & \tau ::= & b \mid \alpha \mid \langle \tau_i{}^{i\in 1..p}\rangle \\
 & \mid P \mid P & & \\
D ::= & J \rhd P & \sigma ::= & \tau \mid \forall\alpha.\sigma \\
 & \mid D \wedge D & & \\
J ::= & x\langle x_i{}^{i\in 1..p}\rangle & A ::= & \emptyset \mid A + (u:\sigma) \\
 & \mid J \mid J & B ::= & \emptyset \mid B + (u:\tau)
\end{array}$$

A process P is either a message, a defining process, or a parallel composition of processes ; a definition D consists of one or several clauses $J \triangleright P$ that associate a guarded process P to a specific message pattern J ; a join-pattern J consists of one or several messages in parallel. We say that the pattern $J = \ldots x\langle z_i \ ^{i\in 1..p}\rangle \ldots$ defines the name x. We note $dv(D)$ for the set of all names that are defined in D.

Processes and definitions are known modulo renaming of bound variables, as substitution performs α-conversion to avoid captures.

A type τ is either a basic type, a type variable, or a message type conveying a fixed number of types ; a type scheme σ may quantify over type variables ; a typing environment A associates type schemes to names, while a simple environment B associates types to names. Given an environment A that already associates a type scheme to a name u, the new environment $A + (u : \sigma)$ is well formed and associates σ to u.

Primitive names $\mathcal{K}$ are given with a primitive typing environment $A_{\mathcal{K}}$ of domain $\mathcal{K}$.

2.2 Typing rules

There are three kinds of typing judgments :

$A \vdash u : \tau$ the name u has type τ in A ;
$A \vdash P$ the process P is well-typed in A ;
$A \vdash D :: B$ the definition D is well-typed in A with types B for its defined names.

The following rules describe valid proofs for our judgments. They are much inspired by the typing rules for the (polyadic) λ-calculus plus `let rec`, the real innovation being the generalization in Def.

(Inst)
$$\frac{u : \forall \alpha_i \ ^{i\in 1..n}. \tau \in A}{A \vdash u : \tau[\tau_i/\alpha_i \ ^{i\in 1..n}]}$$

(Par)
$$\frac{A \vdash P \qquad A \vdash Q}{A \vdash P \mid Q}$$

(Message)
$$\frac{A \vdash u : \langle \tau_i \ ^{i\in 1..n}\rangle \qquad (A \vdash u_i : \tau_i) \ ^{i\in 1..n}}{A \vdash u\langle u_i \ ^{i\in 1..n}\rangle}$$

(Rule)
$$\frac{A + u_{ij} : \tau_{ij} \ ^{i\in 1..n, j\in 1..m_i} \vdash P}{A \vdash x_1\langle u_{1j} \ ^{j\in 1..m_1}\rangle \mid \ldots x_n\langle u_{nj} \ ^{j\in 1..m_n}\rangle \triangleright P :: (x_i : \langle \tau_{ij} \ ^{j\in 1..m_i}\rangle) \ ^{i\in 1..n}}$$

(And)
$$\frac{A \vdash D_1 :: B_1 \qquad A \vdash D_2 :: B_2}{A \vdash D_1 \wedge D_2 :: B_1 \oplus B_2}$$

(Def)
$$\frac{A + B \vdash D :: B \qquad A + \mathrm{Gen}(B, A) \vdash P}{A \vdash \texttt{def}\ D\ \texttt{in}\ P}$$

The rules use the following definitions :

- $B_1 \oplus B_2$ is $B_1 + B_2$, and requires B_1 and B_2 to be equal on $dv(B_1) \cap dv(B_2)$.
- $\mathrm{Gen}(B, A)$ is the generalization of the simple environment B of the form $(x_i : \tau_i)^{\,i \in 1..n}$ with respect to A : let $fv(A)$ be the set $\bigcup_{(x:\sigma) \in A} fv(\sigma)$ where $fv(\sigma)$ contains the free variables of σ ; let $B \backslash x$ be the environment B without the binding for x. Then $\mathrm{Gen}(B, A)$ is $(x_i : \forall\, (fv(\tau_i) - fv(A + B \backslash x_i))\,.\tau_i)^{\,i \in 1..n}$.

2.3 Chemical Semantics

For our type system to be of some use, we must show its consistency with respect to the semantics of the join-calculus.

This semantics is specified as a reflexive chemical abstract machine (RCHAM), as in [2]. The state of the computation is a *chemical soup* $\mathcal{D} \Vdash \mathcal{P}$ that consists of two multisets : active definitions $\mathcal{D}$ and running processes $\mathcal{P}$.

The chemical soup evolves according to two families of rules : *Structural rules* $\rightleftharpoons$ are reversible ($\rightharpoonup$ is heating, $\leftharpoondown$ is cooling) ; they represent the syntactical rearrangement of terms (heating breaks terms into smaller ones, cooling builds larger terms from their components). *Reduction rules* $\longrightarrow$ consume specific processes present in the soup, replacing them by some others ; they are the basic computation steps. In the following, a generic rule will be denoted by the symbol $\rightleftharpoons\!\!\!\!\!\longrightarrow$, and we will write $dv(\mathcal{D})$ for the union $\bigcup_{D \in \mathcal{D}} dv(D)$.

Every rule applies on any matching subpart of the soup. More explicitly, for every rule $\rightleftharpoons\!\!\!\!\!\longrightarrow$, we also have a context rule :

$$\text{(Context)}\quad \frac{\mathcal{D}_1 \Vdash \mathcal{P}_1 \rightleftharpoons\!\!\!\!\!\longrightarrow \mathcal{D}_2 \Vdash \mathcal{P}_2 \qquad (fv(\mathcal{D}) \cup fv(\mathcal{P})) \cap dv(\mathcal{D}_1 \backslash \mathcal{D}_2 \cup \mathcal{D}_2 \backslash \mathcal{D}_1) = \emptyset}{\mathcal{D} \cup \mathcal{D}_1 \Vdash \mathcal{P}_1 \cup \mathcal{P} \rightleftharpoons\!\!\!\!\!\longrightarrow \mathcal{D} \cup \mathcal{D}_2 \Vdash \mathcal{P}_2 \cup \mathcal{P}}$$

A chemical semantics naturally induces a structural equivalence $\equiv$ on terms, defined as the smallest structural congruence that contains $\rightleftharpoons$; this leads to a more classical presentation of the semantics as term rewriting modulo equivalence.

The original machine

In [2, 3], the chemical rules are :

$$\begin{array}{rcll}
\Vdash P_1 \mid P_2 & \rightleftharpoons & \Vdash P_1, P_2 & \textsc{S-Par} \\
D_1 \wedge D_2 \Vdash & \rightleftharpoons & D_1, D_2 \Vdash & \textsc{S-And} \\
\Vdash \mathtt{def}\ D\ \mathtt{in}\ P & \rightleftharpoons & D \Vdash P & \textsc{S-Def} \\
 & & & \\
J \triangleright P \Vdash \varphi(J) & \longrightarrow & J \triangleright P \Vdash \varphi(P) & \text{R-}\beta \\
\Vdash k\langle u_i^{\ i \in 1..p} \rangle & \longrightarrow & \Vdash P & \text{R-}\delta
\end{array}$$

with the side-conditions :

- (S-Def) the names defined in D must not appear anywhere in solution but in the reacting process and definition D and P. This condition is global; in combination with α-renaming it enforces lexical scoping.
- (R-β) $\varphi(\cdot)$ substitute actual names for the received variables in J and P.
- (R-δ) $(u_i\ ^{i\in 1..p}, P) \in \delta_k$, where $\{\delta_k, k \in \mathcal{K}\}$ is a family of primitive relations that map names $u_i\ ^{i\in 1..p}$ to processes P.

We would expect every typing property to be preserved by the structural equivalence, but this is not the case here. The trouble lies in the grouping of definitions that changes outer bound occurrences into recursive ones. Given two definitions D_1 and D_2 such that some names defined by D_1 occur free in D_2, but not the converse, we have

$$\texttt{def}\ D_1\ \texttt{in}\ \texttt{def}\ D_2\ \texttt{in}\ P \equiv \texttt{def}\ D_1 \wedge D_2\ \texttt{in}\ P$$

Unfortunately, the valid typing judgments for the names defined in D_1 and used in D_2 are not the same on each side of the equivalence. Polymorphic typing can be used in the left program and not in the right program. In fact, we run across the classical limitation of typing for mutually-recursive functions.

The restricted machine

To solve this problem, we introduce a variant of the RCHAM that is better suited to our typing purposes. In the new machine, definitions with several clauses are not heated; more specifically, the structural rule S-And disappears and the reduction rule R-β is generalized:

$$\begin{array}{rcrl}
\Vdash P_1 \mid P_2 & \rightleftharpoons' & \Vdash P_1, P_2 & \text{S-Par} \\
\Vdash \texttt{def}\ D\ \texttt{in}\ P & \rightleftharpoons' & D \Vdash P & \text{S-Def} \\
\\
\cdots \wedge J \triangleright P \wedge \cdots \Vdash \varphi(J) & \longrightarrow' & \cdots \wedge J \triangleright P \wedge \cdots \Vdash \varphi(P) & \text{R-}\beta\text{'} \\
\Vdash k\langle u\ ^{i\in 1..p}\rangle & \longrightarrow' & \Vdash P & \text{R-}\delta
\end{array}$$

In the rule R-β' above, $\cdots \wedge J \triangleright P \wedge \cdots$ stands for an active definition that contains the clause $J \triangleright P$. This notation now expresses the commutativity and the associativity of $\wedge$, which were conveyed more explicitly by the structural rule S-And.

In addition, and for every chemical soup $\mathcal{D} \Vdash \mathcal{P}$, we require every name to be defined in exactly one definition of $\mathcal{D}$:

$$\forall D, D' \in \mathcal{D}, dv(D) \cap dv(D') = \emptyset$$

We now relate this restricted machine to the original one. Let us first consider machines that operate on completely diluted solutions (i.e., heating rules cannot apply anymore). There is a straightforward correspondence between chemical solutions of the two formalisms: processes are the same atoms; definitions are equivalent to the clauses that enter into it. Given this equivalence, β-reduction is the same relation in both frameworks. In the general case, structural cooling in the first machine may lead to more programs. However, we still have :

$$(\longrightarrow) \;\subset\; (\longrightarrow') \;\subset\; (\overset{\text{S-And}}{\rightharpoonup})^* \circ (\longrightarrow) \circ (\overset{\text{S-And}}{\leftharpoondown})^*$$

In the following, we use the restricted chemical machine without further discussion. We drop the $'$ notation and write $\longrightarrow$ and $\rightleftharpoons$ for the restricted chemical rules.

2.4 Type-checking solutions

Typing of programs easily extends to chemical solutions. First, we introduce a judgment $A \vdash D$ to state that the assumptions made in A on the names defined in D are the same as if those names had been added in A after typing the definition D. Precisely, we type D in the environment A extended (actually overridden) with new assumptions B that must be exactly the typing environment produced by D as in rule Def ; then, we check that the generalization of B in A is equal to A restricted to $\boldsymbol{dv}(B)$, i.e. $\mathrm{Gen}(B, A)$ is a subset of A. Observe that $A + B$ is also $(A \setminus \boldsymbol{dv}(B)) + B$.

We introduce a new typing judgment $A \vdash \mathcal{D} \Vdash \mathcal{P}$ to state that the chemical solution $\mathcal{D} \Vdash \mathcal{P}$ is well-typed in environment A. This happens when all definitions and all processes are independently well typed in the same environment A :

$$\text{(Multi)}\quad \frac{A + B \vdash D :: B \qquad \mathrm{Gen}(B, A) \subset A}{A \vdash D} \qquad\qquad \text{(Soup)}\quad \frac{\forall P \in \mathcal{P}, A \vdash P \qquad \forall D \in \mathcal{D}, A \vdash D}{A \vdash \mathcal{D} \Vdash \mathcal{P}}$$

Typing chemical solutions simplifies our proofs by avoiding some of the technicalities introduced by the more common formalism of term-rewriting modulo structural equivalence. In particular, the chemistry treats structural rearrangements and proper reductions in the same way. This simplification has already been profitably used in untyped concurrency theory.

3 Correctness of the evaluation

From a quite abstract point of view, let us assume that some evaluation steps of a program P yields a new program P'. Typing and evaluation agree when two facts hold : first, a typing derivation of P' can be constructed from a typing derivation of P. Second, messages present in P' cannot cause “run-time type errors” such as the addition of a string or sending one argument only on a binary name (no type mismatch for primitives, no wrong arity for defined names).

3.1 Assumptions on primitives

For the reduction to be sound, we assume that the primitive reduction relations are consistent with the primitive typing environment $A_{\mathcal{K}}$ introduced in section 2.1. That is, for every typing environment A and for every $k \in \mathcal{K}$, we have:

$$A + A_{\mathcal{K}} \vdash k\langle u_i\ {}^{i \in 1..p}\rangle \text{ and } (u_i\ {}^{i \in 1..p}, P) \in \delta_k \Rightarrow A + A_{\mathcal{K}} \vdash P$$

In particular, the free names of P are either primitives or among the u_i.

3.2 Basic properties for the typing

Lemma 1 (Useless variable) *Let u be a name that is not free in P or D, nor defined in D. Then we have:*

$$\begin{array}{rcl} A \vdash P & \Leftrightarrow & A + (u : \sigma) \vdash P \\ A \vdash D :: B & \Leftrightarrow & A + (u : \sigma) \vdash D :: B \end{array}$$

Lemma 2 (Renaming of type variables) *Let φ be a substitution on type variables. We have:*

$$\begin{array}{rcl} A \vdash P & \Rightarrow & \varphi(A) \vdash P \\ A \vdash D :: B & \Rightarrow & \varphi(A) \vdash D :: \varphi(B) \end{array}$$

We say that a type $\forall \bar{\alpha}.\, \tau$ is *more general* than $\forall \bar{\alpha}'.\, \tau'$ if τ' is of the form $\tau[\bar{\tau''}/\bar{\alpha}]$. This notion lifts to set of assumptions as follows: A' is more general than A if A and A' have the same domain and for each u in their domain, $A'(u)$ is more general than $A(u)$.

Lemma 3 (Generalization) *If $A \vdash P$ and A' is more general than A, then $A' \vdash P$.*

Lemma 4 (Substitution of a name in a term) *If $A + (u : \tau) \vdash P$ and $A \vdash v : \tau$ then $A \vdash P[v/u]$.*

3.3 Subject reduction

Two environments A and A' *agree* when their restrictions to primitive names are equal. We define the relation $\subset$ between RCHAMs as the preservation of typings, that is, $\mathcal{D} \Vdash \mathcal{P} \subset \mathcal{D}' \Vdash \mathcal{P}'$ if for any typing environment A such that $A \vdash \mathcal{D} \Vdash \mathcal{P}$, there exists a typing environment A' such the $A' \vdash \mathcal{D}' \Vdash \mathcal{P}'$ and A and A' agree.

Theorem 1 (subject-reduction) *One-step chemical reductions preserve typings*

<u>Proof</u>: In fact, we prove the stronger property that typing environments also agree on variable names, except maybe on variable names that are defined in either chemical soup but not in both.

That is, for every $\underset{\rightarrow}{\rightleftharpoons}$, let $\mathcal{D} \Vdash \mathcal{P} \underset{\rightarrow}{\rightleftharpoons} \mathcal{D}' \Vdash \mathcal{P}'$ and $A \vdash \mathcal{D} \Vdash \mathcal{P}$. We show that $\mathcal{D}' \Vdash \mathcal{P}'$ is well-typed in an environment A' that possibly differ from A only on $dv((\mathcal{D} \setminus \mathcal{D}') \cup (\mathcal{D}' \setminus \mathcal{D}))$. We prove this property by induction on the number of applications of rule Context in the derivation of the one-step reduction.

Basic case : We first consider the basic case for every reaction rule.

Subcase S-Par : The reduction is $\Vdash P_1 \mid P_2 \rightleftharpoons \Vdash P_1, P_2$.
Heating : Clearly, if $A \vdash P_1 \mid P_2$ then $A \vdash P_1, P_2$ by rules Par and Def.
Cooling : is as easy.

Subcase S-Def : The reduction is $\Vdash \mathtt{def}\ D\ \mathtt{in}\ P \rightleftharpoons D \Vdash P$.
Heating : Let us assume that $A \vdash \mathtt{def}\ D\ \mathtt{in}\ P$, that is, there is a derivation ending with :

$$\frac{A + B \vdash D :: B \qquad A + \mathrm{Gen}(B, A) \vdash P}{A \vdash \mathtt{def}\ D\ \mathtt{in}\ P}\ (\textsc{Def})$$

Clearly, Def and Soup give $A + \mathrm{Gen}(B, A) \vdash D \Vdash P$.
Cooling : Let $A \vdash D \Vdash P$. Then A is of the form $A' + \mathrm{Gen}(B, A')$ and we have both $A' + B \vdash D :: B$ and $A' + \mathrm{Gen}(B, A') \vdash P$. Thus, by Def, $A' \vdash \mathtt{def}\ D\ \mathtt{in}\ P$.

In both cases, the two typing environments agree on primitive names and on names defined both in the solution to the left and to the right of the structural rule.

Subcase R-β : We first assume that D is $J \rhd Q$. Therefore let $A \vdash J \rhd Q \Vdash \varphi(J)$, where J is of the form $x_1\langle \bar{u}_1 \rangle \mid \ldots x_n \langle \bar{u}_n \rangle$. By the rules Def, Soup and Rule, the hypothesis $A \vdash J \rhd Q \Vdash \varphi(J)$ reduces to assume $A = A' + \mathrm{Gen}(B, A')$ and

$$A \vdash \varphi(J) \tag{1}$$
$$A' + B + (\bar{u}_i : \bar{\tau}_i)^{\ i \in 1..n} \vdash P \tag{2}$$

where B is $(x_i : \langle \bar{\tau}_i \rangle)^{\ i \in 1..n}$ and $\mathrm{Gen}(B, A')$ is $(x_i : \forall \bar{\alpha}_i . \bar{\tau}_i)^{\ i \in 1..n}$, where $\bar{\alpha}_i$ is equal to $fv(\bar{\tau}_i) \setminus (fv(A') \cup fv(\bar{\tau}_j)^{\ j \neq i})$.

Observe that the derivation of the judgment (1) must have the shape :

$$\frac{\dfrac{A \vdash x_i : \langle \bar{\tau}'_i \rangle \qquad A \vdash \varphi(\bar{u}_i) : \bar{\tau}'_i}{A \vdash x_i \langle \varphi(\bar{u}_i) \rangle}\ (\textsc{Message}) \qquad i \in 1..n}{A \vdash \varphi(J)}\ (\textsc{Par}) \tag{3}$$

where types $\bar{\tau}'_i$ are type instances $\theta_i(\bar{\tau}_i)$ of $\forall \bar{\alpha}_i . \tau_i$, where θ_i ranges in $\bar{\alpha}_i$. Since generalizable variables never occur in two different bindings, the domains of θ_i's are disjoint and we can define the sum θ of θ_i's for i in $1..n$.

Now, applying the substitution φ, which leaves A' unchanged, to the judgment (2), we get :

$$A' + (x_i : \langle \theta(\bar{\tau}_i) \rangle)^{\ i \in 1..n} + (\bar{u}_i : \theta(\bar{\tau}_i))^{\ i \in 1..n} \vdash P$$

By lemma 3, we can generalize the assumptions of the above judgment as follows :

$$A + (\bar{u}_i : \theta(\bar{\tau}_j))^{\ i \in 1..n} \vdash P$$

This judgment and the hypothesis $A \vdash \varphi(\bar{u}_i) : \bar{\tau}'_i$ of (3) allow to derive, by the name substitution lemma 4, $A \vdash P[\varphi(\bar{u}_j)/\bar{u}_j]$, i.e. $A \vdash \varphi(P)$.

We now consider the general case of a definition $J \triangleright P \wedge D$. By the rules Def and Soup, the hypothesis $A \vdash J \triangleright P \wedge D \Vdash \varphi(J)$ reduces to assuming $A = A' + \mathrm{Gen}(B, A')$ and

$$A' + B \vdash J \triangleright P \wedge D :: B \qquad A \vdash \varphi(J)$$

By the leftmost judgment and the rule And it follows that $B = B' \oplus B''$ and $A' + B' \vdash J \triangleright P :: B'$ and $A' + B'' \vdash D :: B''$. By lemma 1 applied to $A' + B' \vdash J \triangleright P :: B'$ it follows $A' + B \vdash J \triangleright P :: B'$. We reduce to the basic case above by instantiating Rule with this last judgment.

Subcase R-δ: By hypothesis.

Inductive case: We now prove the inductive step uniformly for the context rules. We assume $A \vdash \mathcal{D} \cup \mathcal{D}_1 \Vdash \mathcal{P} \cup \mathcal{P}_1$ and $\mathcal{D}_1 \Vdash \mathcal{P}_1 \rightleftharpoons \mathcal{D}_2 \Vdash \mathcal{P}_2$. By rule Def and Soup, we know that $\mathcal{D}$, $\mathcal{D}_1$, $\mathcal{P}$, and $\mathcal{P}_1$ are all well-typed in A. In particular, $A \vdash \mathcal{D}_1 \Vdash \mathcal{P}_1$. Therefore, by inductive hypothesis, there exists A' such that $A' \vdash \mathcal{D}_2 \Vdash \mathcal{P}_2$, i.e. $A' \vdash \mathcal{D}_2$ and $A' \vdash \mathcal{P}_2$. By inductive hypothesis A and A' agrees modulo names that are defined in $\mathcal{D}_1$ and $\mathcal{D}_2$ but not in both. Let X be such set of names. Then names defined in $\mathcal{D}$ are disjoint from X, by the condition in the premise of the rule (Context). Therefore $A' \vdash \mathcal{D}$ by lemma 1 applied to $A \vdash \mathcal{D}$. Furthermore, the side condition of the rule S-Def also forces $fv(\mathcal{P})$ to be disjoint from $dv((\mathcal{D}_1 \setminus \mathcal{D}_2) \cup (\mathcal{D}_2 \setminus \mathcal{D}_2))$. Thus, we also have $A' \vdash \mathcal{P}$, by lemma 1 applied to $A \vdash \mathcal{P}$. Finally, $A' \vdash \mathcal{D} \cup \mathcal{D}_2 \Vdash \mathcal{P} \cup \mathcal{P}_2$ follows by Soup. ■

3.4 No run-time errors

We state the correctness of a computation from what can be observed from running chemical machines. When ill-formed messages are released in a solution with a consistent set of primitives (see 3.1), there is no reduction that would consume them, so they remain visible, exactly as barbs on free names in an untyped setting. In this case the computation has failed.

Definition 1 A chemical solution $\mathcal{D} \Vdash \mathcal{P}$ has failed when $\mathcal{P}$ contains either:

- A message $k\langle u_i \ ^{i \in 1..n}\rangle$ when no δ-rule applies;
- A message $x\langle u_i \ ^{i \in 1..n}\rangle$ when x is not defined in $\mathcal{D}$, or defined with arity $m \neq n$.

Theorem 2 (Correct computation) *A well-typed chemical machine cannot fail through chemical reduction or equivalence. In particular, a typed program cannot fail.*

Proof: Neither kind of messages of the previous definition can appear in a well-typed chemical soup; chemical typing is preserved by chemical rewriting. ■

4 Type inference

Since types and typing rules are in essence those of ML, our type system also allows for type inference. Precisely, there exists an algorithm that given a soup $\mathcal{D} \Vdash \mathcal{P}$ and a typing environment A_0 that binds the free names of $\mathcal{D}$ and $\mathcal{P}$ with the exception of the active names $\boldsymbol{dv}(\mathcal{D})$, returns a typing environment A of domains $\boldsymbol{dv}(\mathcal{D})$ such that $A_0 \oplus A \vdash \mathcal{D} \Vdash \mathcal{P}$, or fails if no such typing environment exists. Morever, if the algorithms succeeds, then A is principal, that is, for any other typing environment A' of domain $\boldsymbol{dv}(\mathcal{D})$ such that $A_0 \oplus A' \vdash \mathcal{D} \Vdash \mathcal{P}$, then A is more general than A'.

The complete formalization is a straightforward adaptation of the one for ML [1].

5 Functional constructs

In this section, we extend the join-calculus with functions and expressions and we refine the type system accordingly. Such extensions turn the join-calculus into a practical core language that can be seen as a concurrent extension of a small call-by-value functional language with concurrent evaluation and join-call synchronization.

5.1 Programming in the join-calculus

In practice, programmers feel uncomfortable with the non-deterministic behavior of the "print two integer" example of the introduction ; they would often prefer to print `x`, then `y`. The standard trick for enforcing sequential control is to use continuation passing style. Indeed, our implementation provides a synchronous `print_int` primitive that takes two arguments : an integer to be output, and a continuation to be triggered thereafter. This continuation is used for synchronization only ; it carries no argument, and has type $\langle\rangle$. Thus, the type of the synchronous `print_int` is $\langle$`int`$, \langle\rangle\rangle$. The synchronous version of `print_two_ints` also takes an extra continuation argument, and has type $\langle$`int`$,$ `int`$, \langle\rangle\rangle$:

```
def print_two_ints⟨x,y,k⟩ ▷
  def ky⟨⟩ ▷ k⟨⟩ in
  def kx⟨⟩ ▷ print_int⟨y,ky⟩ in
  print_int⟨x,kx⟩
```

The continuation passing style idiom is so common in process calculi that it deserves a convenient syntax that avoids writing explicit continuations (see also [7]). In our setting, continuation arguments are implicit in both primitive names and user-defined names. The synchronous version of "print two integers" becomes :

```
def print_two_ints⟨x,y⟩ ▷
  print_int⟨x⟩ ; print_int⟨y⟩ ; reply to print_two_ints
```

The sequencing operator ";" avoids the definition of explicit continuations inside the body of `print_two_ints`. The final call to continuation is left explicit. This keeps the introduction of synchronous names simple and general, since a join-calculus definition may introduce several synchronous names (and thus several continuations) simultaneously. We write $\texttt{reply}\ u_1, ...u_p\ \texttt{to}\ x$ by analogy to the C `return` instruction. We also provide a sequencing binding $\texttt{let}\ x_1, ...x_p = e\ \texttt{in}\ P$, where e is an expression, i.e. some kind of process with a continuation. More generally, we do not refrain from the temptation of using $\langle\tau_1, \ldots, \tau_q\rangle \rightarrow \langle\tau'_1, \ldots, \tau'_p\rangle$ as a convenient synonym for $\langle\tau_1, \ldots, \tau_q, \langle\tau'_1, \ldots, \tau'_p\rangle\rangle$. Hence, the type of `print_two_ints` can be written $\langle\texttt{int},\texttt{int}\rangle \rightarrow \langle\rangle$.

5.2 Functions as names

Port names can now be used in two different manners: either asynchronously or synchronously. The synchronous invocation of a name u is performed by the new $\texttt{let}\ x_i\ ^{i\in 1..p} = u\langle u_j\ ^{j\in 1..q}\rangle\ \texttt{in}\ P$ construct. The other new construct $\texttt{reply}\ u_i\ ^{i\in 1..p}\ \texttt{to}\ x$ is the asynchronous invocation of the continuation of x. At run-time, it will fire the pending P of a matching $\texttt{let}\ x_i\ ^{i\in 1..p} = x\langle u_j\ ^{j\in 1..q}\rangle\ \texttt{in}\ P$ construct.

$$\begin{array}{ll}
P ::= u\langle u_i\ ^{i\in 1..p}\rangle & \\
\quad \mid \texttt{def}\ D\ \texttt{in}\ P & \tau ::= b \mid \alpha \mid \langle\tau_i\ ^{i\in 1..p}\rangle \\
\quad \mid P \mid P & \quad\ \mid \langle\tau_j\ ^{j\in 1..q}\rangle \rightarrow \langle\tau'_i\ ^{i\in 1..p}\rangle \\
\quad \mid \texttt{let}\ x_i\ ^{i\in 1..p} = u\langle u_j\ ^{j\in 1..q}\rangle\ \texttt{in}\ P & \\
\quad \mid \texttt{reply}\ u_i\ ^{i\in 1..p}\ \texttt{to}\ x & \sigma ::= \tau \mid \forall\alpha.\sigma
\end{array}$$

Patterns, clauses and typing environments are as before. The sequencing operator ";" corresponds to the `let` construct with $p = 0$. There are two additional typing rules for the new constructs:

(Let-Val)

$$\frac{A \vdash u : \langle\tau_j\ ^{j\in 1..q}\rangle \rightarrow \langle\tau'_i\ ^{i\in 1..p}\rangle \qquad (A \vdash u_j : \tau_j)\ ^{j\in 1..q} \qquad A + (x_i : \tau'_i)\ ^{i\in 1..p} \vdash P}{A \vdash \texttt{let}\ x_i\ ^{i\in 1..p} = u\langle u_j\ ^{j\in 1..q}\rangle\ \texttt{in}\ P}$$

(Reply)

$$\frac{A \vdash x : \langle\tau_j\ ^{j\in 1..q}\rangle \rightarrow \langle\tau'_i\ ^{i\in 1..p}\rangle \qquad (A \vdash u_i : \tau'_i)\ ^{i\in 1..p}}{A \vdash \texttt{reply}\ u_i\ ^{i\in 1..p}\ \texttt{to}\ x}$$

The typing rules guarantee that synchronous and asynchronous invocations on the same name do not mix. Moreover, an user-defined name x must be invoked synchronously when its definition includes type consistent occurrences of the $\texttt{reply}\ u_i\ ^{i\in 1..p}\ \texttt{to}\ x$ construct.

Therefore, we give names an asynchronous or synchronous status. Name status may be determined by typing, provided the following agreement: every name whose synchronous usage is not detected by the type inference system is considered asynchronous. In the following, we simply write f, instead of x, for synchronous names.

5.3 A typed CPS encoding

As usual for process calculi, we translate functional names back to the initial join-calculus. The translation applies to type correct programs and once synchronous names have been identified. Given a synchronous name f, we introduce the fresh name κ_f for its continuation ; we also use the reserved name κ for intermediate continuations generated while translating `let`s. The call-by-value translation is :

$$\begin{array}{rcll}
f\langle u_i {}^{i \in 1..p}\rangle & \overset{def}{=\!=} & f\langle u_i {}^{i \in 1..p}, \kappa_f\rangle & \text{(in join-patterns } J\text{)} \\
\texttt{reply}\, u_i {}^{i \in 1..p}\, \texttt{to}\, f & \overset{def}{=\!=} & \kappa_f \langle u_i {}^{i \in 1..p}\rangle & \text{(in guarded processes } P\text{)} \\
\texttt{let}\, x_i {}^{i \in 1..p} = f\langle u_j {}^{j \in 1..q}\rangle \,\texttt{in}\, P & \overset{def}{=\!=} & \texttt{def}\, \kappa\langle x_i {}^{i \in 1..p}\rangle \triangleright P \,\texttt{in}\, f\langle u_j {}^{j \in 1..q}, \kappa\rangle &
\end{array}$$

$$\langle \tau_j {}^{j \in 1..q}\rangle \rightarrow \langle \tau'_i {}^{i \in 1..p}\rangle \quad \overset{def}{=\!=} \quad \langle \tau_j {}^{j \in 1..q}, \langle \tau'_i {}^{i \in 1..p}\rangle\rangle$$

The two additional typing rules, once their operands have been translated, are derived from the type system of section 2 :

(translated Let-Val)

$$\frac{A \vdash f : \langle \tau_j {}^{j \in 1..q}, \langle \tau'_i {}^{i \in 1..p}\rangle\rangle \qquad (A \vdash u_j : \tau_j)\,{}^{j \in 1..q} \qquad A + (x_i : \tau'_i)\,{}^{i \in 1..p} \vdash P}{A \vdash \texttt{def}\, \kappa\langle x_i {}^{i \in 1..p}\rangle \triangleright P \,\texttt{in}\, f\langle u_j {}^{j \in 1..q}, \kappa\rangle}$$

(translated Reply)

$$\frac{A \vdash f : \langle \tau_j {}^{j \in 1..q}, \langle \tau'_i {}^{i \in 1..p}\rangle\rangle \qquad (A \vdash u_i : \tau'_i)\,{}^{i \in 1..p}}{A \vdash \kappa_f \langle u_i {}^{i \in 1..p}\rangle}$$

5.4 A join-calculus-based language

A complete syntax for processes and expressions is :

$$\begin{array}{llcll}
P ::= & u\langle E_i {}^{i \in 1..p}\rangle & \qquad & E ::= & u \\
 & \mid \texttt{def}\, D \,\texttt{in}\, P & & & \mid u\langle E_i {}^{i \in 1..p}\rangle \\
 & \mid P \mid P & & & \mid \texttt{def}\, D \,\texttt{in}\, E \\
 & \mid \texttt{let}\, x_i {}^{i \in 1..p} = E \,\texttt{in}\, P & & & \mid \texttt{let}\, x_i {}^{i \in 1..p} \triangleright E \,\texttt{in}\, E \\
 & \mid \texttt{reply}\, E_i {}^{i \in 1..p}\, \texttt{to}\, x & & &
\end{array}$$

Clauses and definitions are as before.

Again, expressions are only a convenient syntactic sugar, which can be removed. This new translation amounts to introducing explicit bindings of the kind of the previous section for all subexpressions, nested calls being translated top-down, left-to-right.

$$\begin{array}{rcl}
\texttt{reply}\, E_i {}^{i \in 1..p}\, \texttt{to}\, f & \overset{def}{=\!=} & \kappa_f \langle E_i {}^{i \in 1..p}\rangle \text{ (in guarded processes } P\text{)} \\
u\langle E_i {}^{i \in 1..p}\rangle & \overset{def}{=\!=} & (\texttt{let}\, x_i = E_i \,\texttt{in})\,{}^{i \in 1..p} u\langle x_i {}^{i \in 1..p}\rangle \\
\texttt{let}\, x = u \,\texttt{in}\, P & \overset{def}{=\!=} & P\{x/u\} \\
\texttt{let}\, x_i {}^{i \in 1..p} = f\langle E_j {}^{j \in 1..q}\rangle \,\texttt{in}\, P & \overset{def}{=\!=} & \texttt{def}\, \kappa\langle x_i {}^{i \in 1..p}\rangle = P \,\texttt{in}\, f\langle E_j {}^{j \in 1..q}, \kappa\rangle
\end{array}$$

In practice, we introduce new typing judgments for expressions ($A \vdash E : \tau_i{}^{i\in 1..p}$), along with new typing rules. The typing rules for expressions are derived from the previous ones and are omitted.

5.5 A comparison with functional types

If we remove join-composition in patterns and parallel-composition in processes from our language, we get a polyadic functional kernel similar to core-ML : both the reductions and the typing rules do correspond. Let us consider in detail how we would translate the `let` binder of ML. According to the `let`-bound expression, there are two cases with distinct typing properties. When the syntax suffices to identify functions either directly or as aliases, we use a generalizing definition :

$$\begin{aligned}
[\![\texttt{let}\ f(x) = e_1\ \texttt{in}\ e_2]\!] &= \texttt{def}\ f\langle x\rangle \triangleright \texttt{reply}\ e_1\ \texttt{to}\ f\ \texttt{in}\ e_2 \\
[\![\texttt{let}\ g = \texttt{let}\ f(x) = e_1\ \texttt{in}\ f\ \texttt{in}\ e_2]\!] &= \texttt{def}\ f\langle x\rangle \triangleright \texttt{reply}\ x\ \texttt{to}\ f\ \texttt{in}\ e_2[f/g]
\end{aligned}$$

For other values (e.g. function calls), we use a continuation message to convey the result, which forces this result to be monomorphic. This restricts polymorphism to syntactic values and is thus equivalent to Wright's proposal for ML [12] :

$$[\![\texttt{let}\ x = f(u)\ \texttt{in}\ e_2]\!] = \texttt{def}\ \kappa\langle x\rangle \triangleright e_2\ \texttt{in}\ f\langle u, \kappa\rangle$$

Typing side-effects

The language as a whole is more expressive than ML ; it provides support for general, concurrent programming, including imperative constructs and side-effects as messages. For instance, reference cells need not be taken as primitives ; they are programmable in the join-calculus using the following definition :

```
def ref⟨x⟩ ▷
  def get ⟨⟩ | state⟨x⟩ ▷ state⟨x⟩ | reply x to get
  and set⟨y⟩ | state⟨x⟩ ▷ state⟨y⟩ | reply to set
  in  state⟨x⟩ | reply get,set to ref in ...
```

Here, the `state` is kept local and, more importantly, both methods `get` and `set` are returned in the same message. Since only one name `ref` has been defined, its type $\langle\alpha \rightarrow \langle\langle\rangle \rightarrow \langle\alpha\rangle, \langle\alpha\rangle \rightarrow \langle\rangle\rangle\rangle$, can obviously be generalized. And `ref` can be used polymorphically :

```
let g1,s1 = ref⟨'hello'⟩ in
let g2,s2 = ref⟨3⟩ in ...
```

More generally, join-calculus definitions may describe protocols that involve sophisticated synchronization of numerous methods and/or partial states, but this is largely independent of the typing, as long as side-effects are tracked using the sharing of type variables.

This is in contrast with the classical approach in ML, where references are introduced in a “pure” language as dangerous black boxes that cannot be given polymorphic types, and that communicate with a global store by magic. In [13], references are introduced as local stores that can be extruded. This is slightly closer to the join-calculus, but again references are a new special construct. If required, the store can still be identified as some part of the chemical machine, that consists of the instances of cell definitions on the left-hand-side, and of their state messages on the right-hand-side. The approach taken in the join-calculus is uniform and, by the way, it allows to type at least as much as ML with references.

6 Related works for concurrent languages

In the area of name-passing process calculi, the first step was taken by Milner in 1991 [5]. Milner introduced an improvement of the π-calculus, called *polyadic π-calculus*, where channels are allowed to carry tuples of messages. Polyadicity naturally supports a concept of “sorting”, which is in our view a humble word for typing. In the context of polyadic π-calculus, maintaining the type discipline enforces channels to always carry tuples of the same length and nature.

The first extension of Milner’s system has been undertaken by Pierce and Sangiorgi. They distinguish between input-only, output-only, and input-output channels. This extension naturally leads to recursive types with subtyping [6]. Since then more and more elaborate extensions have then been proposed and experimented, mostly around the `Pict` language, a strongly-typed implementation of the π-calculus with support for functions and objects [7, 10]. Recently, a further extension captures linearity information in channel types [4]. This provides a finer account on communication patterns, and static type inference leads to a more efficient compilation.

The type systems of all these authors are usually more sophisticated than ours. Some of this sophistication is due to the complexity of the π-calculus semantics and is thus irrelevant in our case. Nevertheless, sophisticated static analysis such as deadlock or linearity analysis would be useful in an optimizing join-calculus compiler. We chose not to integrate such high-level analyses in the basic type system.

The basic theory of polymorphic extensions of Milner’s sort discipline for π-calculus has been developed by Turner in his PhD thesis [10]. We recall that Turner’s polymorphism is *explicit*: inputs and outputs are always annotated with sorts. For example, the π-calculus process $\overline{x}[y, z] \mid x[u, w].\,\overline{u}[w]$ is tagged as follows:

$$\overline{x}[\mathtt{int}; y, z] \mid x[\alpha; u :\uparrow \alpha, w : \alpha].\,\overline{u}[\alpha; w]\ .$$

Consequently, explicit abstraction and application of types are interleaved with communication: in the above example the sort α in the output $\overline{u}[\alpha; w]$ depends on the message received on the channel x. This commitment to explicit polymorphism in π-calculus follows from the absence of a place where sort generalization may occur.

Typing à la ML for concurrent languages is not new. Proposals have been defined for languages which combine functional and concurrent primitives (among the others, Concurrent ML [8] and Facile [9]). An analogous approach has recently been taken by Vasconcelos for an extension of π-calculus with agent names [11]. In these languages channels are always monomorphic, and polymorphism is only allowed under functional abstractions. This enables to parameterize processes by arguments of different types. However, two processes can never communicate values of different types on the same channel, which restricts the expressiveness of the language. In particular, it is impossible to implement polymorphic services. As a simplified example, consider a computing server :

```
def run⟨f,x,r⟩ ▷ f⟨x,r⟩ in ...
```

This defines a channel `run` of type $\langle\langle\alpha,\langle\beta\rangle\rangle,\alpha,\langle\beta\rangle\rangle$ to which expensive requests can be sent together with a channel to receive the result (in the distributed join calculus, the location of the server would also be passed to `f` so that `f` can choose to migrate to the server before intensive computing.)

There is apparently no way to define such a service in CML, Facile, or the language proposed in [11]. This limitation is in fact known, and is one of the motivations for adding explicit existential types in ML.

7 Conclusion

We have typed the join-calculus using traditional parametric polymorphism. Thereby, we demonstrate that successful concepts and techniques now familiar in functional programming carries over to concurrent programming.

This experience strengthens our confidence both in the join-calculus and in parametric polymorphism. The join-calculus is a practical concurrent programming language because it support a simple, convenient and well established typing paradigm. ML polymorphism is not bound to ML ; it can sustain significant changes in the language semantics, provided lexical scoping is maintained and generalization points are clearly identified.

References

[1] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings on Principles of Programmining Languages*, pages 207 – 212, 1982.

[2] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, 1996.

[3] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, 1996. LNCS 1119.

[4] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linear types and pi-calculus. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, 1996.

[5] R. Milner. The polyadic π-calculus : a tutorial. In Bauer, Brawer, and Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer Verlag, 1993.

[6] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, pages 187 – 215, 1993.

[7] B. Pierce and D. Turner. Pict : a programming language based on the pi-calculus, 1995. To appear.

[8] J. H. Reppy. Concurrent ML : Design, application and semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, pages 165 – 198, 1992. LNCS 693.

[9] B. Thomsen. Polymorphic sorts and types for concurrent functional programs. Technical Report ECRC-93-10, European Computer-Industry Research Center, Munich, Germany, 1993.

[10] D. N. Turner. *The π-calculus : Types, polymorphism and implementation.* PhD thesis, LFCS, University of Edinburgh, 1995.

[11] V. T. Vasconcelos. Predicative polymorphism in the π-calculus. In *Proceedings of 5th Conference on Parallel Architectures and Languages (PARLE 94)*, 1994. LNCS.

[12] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93-200, Rice University, February 1993.

[13] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1) :38–94, 1994.