A Calculus of Tagged Types, with applications to process languages

Bernard Berthomieu and Camille le Moniès de Sagazan

LAAS / CNRS, 7, Avenue du Colonel Roche, 31077 Toulouse Cedex, France
Tel: (+33) 61 33 62 00, Fax: (+33) 61 33 64 11, Telex: 520930 F (LAASTSE)
e-mail: Bernard.Berthomieu@laas.fr, Camille.le.Monies.de.Sagazan@laas.fr

Abstract: Tagged Types encode some families of types indexed by labels. Like Wand's row types, they make use of specific variables from which new fields can be extracted, and like Rémy's record types, the types introduced in these fields can be parameterized. However, the logical and technical treatment of tagged types is original; it is claimed that they combine the intuitiveness of row types with most of the expressiveness of record types. Their use is illustrated by the design of polymorphic typing systems for various process calculi extending CCS. Processes are assigned types which associate polymorphic types with all communication labels.

1. Introduction

1.1 Types for tagged expressions

Typing record expressions or some behavior expressions, or inferring signatures in module systems, require collections of types associated with labels. A number of proposals have been done to give such structures some extensions capabilities. Wand introduced *row* variables in [20] to type records with partial knowledge of their fields. Rémy [13] subsequently used infinitary products and proposed a convenient polymorphic type system for record expressions. A connection between the two formalisms is made clear in [21] where row types are presented as a handy notation for some of Rémy's record types.

In row types notation, a function taking a record as argument and extracting the content of its field labelled a has type $\{\mathtt{a}: P(\alpha)\}\Delta \rightarrow \alpha$ where encapsulation by P means that field a is required and the row variable Δ implicitly associates some type variable α_b with every label $\mathtt{b}\neq\mathtt{a}$. In Rémy's notation, it has type $\{\mathtt{a}: P(\alpha); \beta\} \rightarrow \alpha$, where β is a "pattern" for the types in other fields than that labelled a.

Record types and row types may be read as total mappings, assigning types to all labels. Consistently with Rémy's treatment, row variables may be read as partial mappings. The row type $\{\mathtt{a}: P(\mathrm{int})\}\Delta$, for instance, associates a type with every possible record label but, in that context, the row variable Δ does not assign any type to label a since one is already assigned to it. This interpretation implies a well-formedness condition for types excluding types like e.g. $\Delta \rightarrow \{\mathtt{a}:\mathrm{int}\}\Delta$.

The Tagged Types we introduce use row variables, but these are interpreted as total mappings; well-formedness of types becomes a simple grammatical constraint. Label replications (as in {p:int}{p:β}Ω, obtained from {p:int}Δ by substituting Δ) are handled by considering that a field supersedes any inner fields bearing the same label. In addition, row variables are associated with type schemes used as patterns for the types implicitly assigned by the variables; this gives tagged types most of the flexibility of Rémy's record types. From an axiomatization of their equality, we derive unification algorithms that extend in a simple way the standard techniques.

1.2 Typing process calculi

Tagged types were developed to typecheck parallel programs written in LCS [3], a language combining ML and an higher order version of the Calculus of Communicating System [9] with behavior passing and parametric channels.

There are basically two classes of process programming languages, according to whether or not communication channels are considered first-order values.

Languages in the former class include the π-calculus [10], CML [15], FACILE [5] and Plain CHOCS [17]. In these languages, channels are typically typed like locations in a store; processes are merely checked in an environment assigning types to channels and other variables, and are given a trivial *process* type (see e.g. [7]). Alternatively, [12] annotate functional types with behavioral informations from which various static analyses, including typing, can be performed.

Calculi and languages in the latter class include CHOCS [16], TPL [11] and LCS [3]. The "channels as locations" analogy would give here a poor typing system, rejecting many useful programs. A better approach is to see processes as records of channels and to assign each process the set of types expected on the communication labelsit uses. This is basically the approach of [11].

Computing statically that set of labels is not always possible, in particular in languages allowing process-passing or permitting to define process combinators. Such features can be handled by making the type of a process assign a type information to all possible communication labels, and not only to those used by the process; such process types are conveniently encoded by tagged types. This is the approach taken for LCS; it bears strong similarities with the way record expressions are typed in [21] or [13].

1.3 Plan of the paper

Tagged Types are introduced in Section 2, including unification techniques. Section 3 illustrates their use for typing CCS programs and various higher order extensions. We conclude with some comparisons with related work. Most of the results are given here without proofs, these will be included in a longer version of this paper. More details can be found in [2], an earlier attempt at formalizing tagged types, or in the second author's forthcoming thesis [6].

2. Tagged Types

2.1 The language of tagged types

Given a countable set of labels $\Sigma = \{p, q, \ldots\}$, *types* τ, of which the *tagged types* ρ are a subset and *type schemes* σ obey the following grammar:

τ	::=	$\alpha, \beta, \gamma, \ldots$	plain type variables
	|	$a, b, c, \ldots$	plain constant types
	|	$\tau \rightarrow \tau$	function type
	|	ρ	tagged types
ρ	::=	$\Delta^{\sigma}, \Theta^{\sigma'}, \ldots$	tagged type variables, σ, σ' closed
	|	$\nabla^{\mu}, \nabla^{\mu'}, \ldots$	tagged type constants, μ, μ' monotypes
	|	$\{p\colon \tau\}\rho$	field prefixing, for each $p \in \Sigma$
σ	::=	$\tau \mid \forall\{\alpha \mid \Delta^{\sigma'}\}.\ \sigma$	type schemes

The types are classified into sorts; infinitely many variables are available at each sort. Plain variables and plain constants have the *plain* sort $\mathcal{U}$. A tagged variable Δ^{σ} or tagged constant ∇^{σ} has the *tagged* sort $\mathcal{T}(\sigma)$; each closed type scheme defines a tagged sort. Sort assignment for the other types will soon be made precise. A *monotype* is a type in which no variable occurs (of any sort). Though this is not essential, it is assumed that tagged variables have polymorphic sorts (i.e. the closed type scheme σ in Δ^{σ} contains at least one variable).

Tagged types have a number of *fields* prefixing a tagged variable or tagged constant (called the *extension*); each field associates a type with a label. The type $\{p_1{:}\tau_1\}\ldots\{p_k{:}\tau_k\}\rho$ is also written $\{p_1{:}\tau_1, \ldots, p_k{:}\tau_k\}\rho$ when type ρ has no fields and labels p_l are pairwise distinct. Sort annotations are omitted for tagged variables having sort $\mathcal{T}(\forall\alpha.\alpha)$: Δ, Θ, stand for the variables $\Delta^{\forall\alpha.\alpha}$, $\Theta^{\forall\alpha.\alpha}$.

As usual, type schemes are types possibly quantified at the outermost. $\bar{\tau}$ denotes the closure of type τ, obtained by quantifying all of its variables.

Finally, we restrict here the possible tagged sorts to those not including themselves any tagged variables. This is for the sake of simplicity; the treatment developed in the following sections is easily extended to higher order tagged sorts.

2.2 Substitutions

[ ] is the identity substitution; $[\tau_1/\alpha_1^s,\ldots,\tau_k/\alpha_k^{s'}]$ replaces variables $\alpha_1^s,\ldots,\alpha_k^{s'}$ by types $\tau_1,\ldots,\tau_k$, respectively; $S_2 o S_1$ is the composition of S_1 and S_2. $\leq$ is the substitution instance preorder ($\tau \leq \tau' \Leftrightarrow \exists\, S.\ \tau = S\,\tau'$), and $\equiv$ is the associated equivalence relation ($\tau \equiv \tau' \Leftrightarrow \tau \leq \tau' \wedge \tau' \leq \tau$).

To define substitutions, it is convenient to cast our types into the framework of order-sorted languages: Terms in an order sorted language are classified into sorts, and the set of sorts is equipped with a partial ordering $\subseteq$ (often called sort inclusion). Substitutions are restricted to those preserving sorts: For any substitution S and variable v, we must have $\mathrm{Sort}(S\ v) \subseteq \mathrm{Sort}(v)$.

The sort membership rules for variables and constants were explained in Section 2.1. Function types have the plain sort $\mathcal{U}$. A tagged type $\{p{:}\tau\}\rho$ has sort $\mathcal{T}(\sigma')$ when $\mathcal{T}(\bar{\tau}) \subseteq \mathcal{T}(\sigma')$ and $\mathcal{T}(\sigma) \subseteq \mathcal{T}(\sigma')$, where $\mathcal{T}(\sigma)$ is the sort of type ρ. Relation $\subseteq$ includes all pairs $\mathcal{T}(\sigma) \subseteq \mathcal{U}$, for any σ, and all pairs $\mathcal{T}(\bar{\tau}) \subseteq \mathcal{T}(\bar{\tau}')$ with $\tau \leq \tau'$.

In other words, a plain variable may be substituted by any type while a tagged variable of sort $\mathcal{T}(\bar{\tau})$ may only be substituted by tagged types in which the type in each field and the exponent of the extension are instances of τ. E.g. let $\tau = \{p{:}\alpha\}\Delta^{\forall\delta\beta.\delta\to\beta}$; then the types $\{p{:}\alpha\}\{q{:}\lambda\to\mu\}\Theta^{\forall\delta\beta.\delta\to\beta}$ and $\{p{:}e\}\nabla^{b\to c}$ are substitution instances of τ, but neither $\{p{:}\alpha\}\Delta^{\forall\beta.\beta}$ nor $\{p{:}\alpha\}(\gamma\to\delta)$ are.

A *renaming* is a substitution mapping variables to variables and injective on its domain; renamings are written R, R', etc. We have $\tau \equiv \tau'$ iff $\exists\, R.\ \tau = R\,\tau'$. A substitution is *nonexpansive* when it maps tagged variables to tagged variables; nonexpansive substitutions are written N, N', etc.

2.3 Equality

Models of tagged types

To avoid a semantics of tagged types tied to a specific programming notation, we define an extensional equality that we assume valid in all models of closed type schemes. Equality of closed type schemes ($=_g$) is defined from their monotype instances (Π, Π' are tagged monotypes):

$$\sigma =_g \sigma' \quad \Leftrightarrow \quad (\forall\mu\leq\sigma \Rightarrow \exists\mu'\leq\sigma'.\ \mu =_g \mu') \wedge (\forall\mu'\leq\sigma' \Rightarrow \exists\mu\leq\sigma.\ \mu =_g \mu')$$

$$a =_g a \qquad \text{(for each plain constant } a\text{)}$$

$$\mu_1\to\mu_1' =_g \mu_2\to\mu_2' \quad \text{if} \quad \mu_1 =_g \mu_2 \ \wedge\ \mu_1' =_g \mu_2'$$

$$\Pi =_g \Pi' \quad \text{if} \quad \forall p\in\Sigma.\ \Pi\,(p) =_g \Pi'\,(p)$$

with Π (p) obtained by:

$$(\{p{:}\mu\}\Pi)\,(p) = \mu$$

$$(\{q{:}\mu\}\Pi)\,(p) = \Pi\,(p) \qquad (q\neq p)$$

$$\nabla^{\mu}\,(p) = \mu$$

Monotype tagged types are seen as mappings; ∇^{μ} is a constant mapping; $\{p{:}\mu\}\Pi$ stands for the mapping obtained from Π by replacing its (p,_) pair by (p,μ). We axiomatize in the sequel equality $=_g$.

~-equality

Relation ~ is the smallest congruence including identity and the pairs from:

(*permutation*)	$\{p{:}\ \tau\}\{q{:}\ \tau'\}\rho \sim \{q{:}\ \tau'\}\{p{:}\ \tau\}\rho$	(for any p,q: $p \neq q$)
(*pruning*)	$\{p{:}\ \tau\}\{p{:}\ \tau'\}\rho \sim \{p{:}\ \tau\}\rho$	(for any p)
(μ-*expansions*)	$\nabla^{\mu} \sim \{p{:}\ \mu\}\nabla^{\mu}$	(for any p)

Fields may appear in any order. A field may be removed when it follows another field bearing the same label; such fields are said *hidden*. The last axiom expresses how to extract a field out of a tagged constant.

~-equality is preserved by substitutions. $|\tau|$ is the *normal form* of type τ, obtained from τ by removing its hidden fields. $\mathcal{V}(\tau)$ is the set of variables occurring in $|\tau|$.

Expansion equivalence

We would like to express some extensionality axiom for tagged variables, similar to the one above for tagged constants. For this, we need some method to control introduction of variables:

An *expansion component* is a substitution $[\{p{:}\tau'\}\Delta^{\bar{\tau}}/\Delta^{\bar{\tau}}]$, with $\tau' \equiv \tau$ and $(\mathcal{V}(\tau)\cup\{\Delta^{\bar{\tau}}\})\cap\mathcal{V}(\tau')=\varnothing$. Given a set W of *protected variables*, an *expansion basis away from* W is a set of substitutions such that (1): It holds the empty substitution and exactly one component $[\{p{:}\tau'\}\Delta^{\bar{\tau}}/\Delta^{\bar{\tau}}]$ for each pair $(\Delta^{\bar{\tau}},p)$, (2): For any two components $[\{p{:}\tau\}\Delta^{\sigma}/\Delta^{\sigma}]$ and $[\{q{:}\tau'\}\Theta^{\sigma'}/\Theta^{\sigma'}]$ in the set, we have $\mathcal{V}(\tau)\cap\mathcal{V}(\tau')=\varnothing$ and $(\mathcal{V}(\tau)\cup\mathcal{V}(\tau'))\cap W=\varnothing$ and (3): it is closed by composition of substitutions.

For each W, Ξ^{W} denotes the union of all expansion basis away from W; the elements of Ξ^{W} are called *expansions away from* W.

The *expansion preorder* $\lesssim_{\eta}$, and W-*expansion equivalence* are defined by:

$$\tau \lesssim_{\eta} \tau' \quad\Leftrightarrow\quad \exists W, X\in\Xi^{W}.\ \mathcal{V}(\tau')\subset W \wedge \tau \sim X\,\tau'$$
$$\tau \sim_{\eta}^{W} \tau' \quad\Leftrightarrow\quad \exists\tau''.\ \mathcal{V}(\tau'')\subset W \wedge \tau \lesssim_{\eta} \tau'' \wedge \tau' \lesssim_{\eta} \tau''$$

$\tau \lesssim_{\eta} \tau'$ means that τ may be obtained from τ' by some possibly empty expansion protecting its relevant variables. The types introduced in any fields made explicit follow from the sorts of the variables expanded. The definition of Ξ^{W} prevents any expansion to introduce the same variable twice, or variables in set W.

$\sim_\eta^W$ is an equivalence relation, but not a congruence. A type is seen equivalent with any type obtained from it by applying an expansion protecting its variables.

Instances and $\cong$*-equivalence*

The substitution instance preorder $\leq$ is too restrictive for our purpose; we want in addition to consider types up to ~ and expansions. We define thus the coarser preorder $\lesssim$ (read *less or equally general*) and the associated equivalence relation $\cong$ (read *equally general*) by:

$$\tau \lesssim \tau' \quad \Leftrightarrow \quad \exists\, \tau_1, \tau_2.\ \tau \gtrsim_\eta \tau_1 \leq \tau_2 \lesssim_\eta \tau'$$
$$\tau \cong \tau' \quad \Leftrightarrow \quad \tau \lesssim \tau' \wedge \tau' \lesssim \tau$$

Example 1: Consider the following types (all tagged variables have sort $\mathcal{T}(\forall\alpha.\alpha)$):

(1) $\{p{:}\alpha\}\{s{:}\gamma\}\Delta \to \{p{:}\beta\}\{s{:}\gamma\}\Delta$ (2) $\{p{:}\alpha\}\Delta \to \{p{:}\beta\}\Delta$ (3) $\{p{:}\alpha\}\Delta \to \Delta$

(4) $\{p{:}\beta\}\{s{:}\lambda\}\Theta \to \{p{:}\omega\}\{s{:}\lambda\}\Theta$ (5) $\{p{:}\beta\}\Theta \to \{p{:}\omega\}\Theta$ (6) $\Theta \to \{p{:}\omega\}\Theta$

We clearly have (1) $\lesssim_\eta$ (2) $\lesssim_\eta$ (3) and (4) $\lesssim_\eta$ (5) $\lesssim_\eta$ (6), but neither (3) $\lesssim_\eta$ (6) nor (6) $\lesssim_\eta$ (3). However, we have (3) $\lesssim$ (6) and (6) $\lesssim$ (3) since (3)$\gtrsim_\eta$(2)$\leq$(5)$\lesssim_\eta$(6) and (6)$\gtrsim_\eta$(5)$\leq$(2)$\lesssim_\eta$(3); all six types are equivalent by $\cong$. None of them are related by ~.

$\lesssim$ and $\cong$ can be characterized as follows, in terms of substitutions (*R* is a renaming and *N* is a *nonexpansive* substitution):

$$\tau \lesssim \tau' \quad \Leftrightarrow \quad \exists\, W, X \in \Xi^W, N.\ \mathcal{V}(\tau) \cup \mathcal{V}(\tau') \subset W \wedge X\,\tau \sim (N \circ X)\,\tau'$$
$$\tau \cong \tau' \quad \Leftrightarrow \quad \exists W, X \in \Xi^W, R.\ \mathcal{V}(\tau) \cup \mathcal{V}(\tau') \subset W \wedge (R \circ X)\,\tau \sim X\,\tau'$$

Equality of type schemes and canonical forms

Equality of closed type schemes is defined by $\bar{\tau} \cong \bar{\tau}' \Leftrightarrow \tau \cong \tau'$. That equality is sound versus the extensional equality $=_g$ discussed earlier: $\sigma \cong \sigma'$ implies $\sigma =_g \sigma'$. It is conjectured that the converse implication is also true.

Several canonical forms can be defined for closed type schemes. They all follow from the fact that the number of types in normal form which are related by $\cong$ to some type, and are maximal by preorder $\lesssim_\eta$, is finite (modulo a renaming of variables and a permutation of fields). For the types in Example 1, for instance, there are exactly two such maximal classes, including types (3) and (6), respectively. Any total ordering on maximal classes defines a canonical class; the greatest lower bound by $\lesssim_\eta$ of these maximal classes defines another canonical class. In Example 1, that latter class would hold types (2) and (5).

2.4 Unification

~-unification: A substitution U is a *~-unifier* of (τ_1,τ_2) iff we have $U\,\tau_1 \sim U\,\tau_2$. In addition, U is a *~-most general unifier* (*~-mgu* for short) iff for any other ~-unifier V of (τ_1,τ_2), we have $V\,\tau_1 \lesssim U\,\tau_1$.

If U and U' are two ~-mgus of (τ_1,τ_2), then we must have $U\,\tau_1 \cong U'\,\tau_1$.

Theorem: There is an algorithm that returns for any types τ and τ' a ~-principal ~-unifier, or indicates failure if these types are not ~-unifiable.

We give a proof sketch by reducing ~-unification to an order-sorted variant of the standard first order unification problem:

In what follows, $S\,\tau_i \sim S'\tau_i$ stands for $S\,\tau_1 \sim S'\,\tau_1 \wedge S\,\tau_2 \sim S'\,\tau_2$ and all expansions are assumed away from the variables occurring in the types to which they are applied, as well as away from all variables in the surrounding context when unification is used as part of a more general process,

From the characterization of preorder $\lesssim$, finding a ~-mgu of (τ_1,τ_2) is equivalent to finding a nonexpansive substitution N, and an expansion X such that :

(i) $(N\text{o}X)\,\tau_1 \sim (N\text{o}X)\,\tau_2$

(ii) $\forall N',X'.\,(N'\text{o}X')\,\tau_1 \sim (N'\text{o}X')\,\tau_2 \Rightarrow \exists X'',W.\,(X''\text{o}N'\text{o}X')\,\tau_i \sim (W\text{o}N\text{o}X)\,\tau_i$

If such an N and X are found, then $(N\text{o}X)$ is a ~mgu of (τ_1,τ_2). That problem can be shown equivalent to finding N and X such that:

(i') $(N\text{o}X)\,\tau_1 \sim (N\text{o}X)\,\tau_2$

(ii') $\forall N'.\,(N'\text{o}X)\,\tau_1 \sim (N'\text{o}X)\,\tau_2 \Rightarrow \exists W.\,(N'\text{o}X)\,\tau_i \sim (W\text{o}N\text{o}X)\,\tau_i$

(iii) $\forall N',X'.\,N'$ ~-mgu of $(X'\text{o}X\,\tau_1, X'\text{o}X\,\tau_2)$
$\qquad\Rightarrow \exists X''.\,(N'\text{o}X'\text{o}X)\,\tau_i \sim (X''\text{o}N\text{o}X)\,\tau_i$

A sufficient condition to guarantee the commutation property (iii) is that expansion X makes all tagged variables in (τ_1,τ_2) bear fields for exactly the same set of labels. Further, given such an expansion X, $(X\,\tau_1, X\,\tau_2)$ is necessarily equal by ~ to some pair (τ_1', τ_2') holding no hidden fields, in which tagged variables and constants are all preceded by fields for the same set of labels, and these fields occur in the same label-order. Since ~ is preserved by substitutions and $\tau_i' \sim X\,\tau_i$, an equivalent formulation of the previous problem is to find a nonexpansive substitution N such that:

(i'') $N\,\tau_1' = N\,\tau_2'$

(ii'') $\forall N'.\,N'\,\tau_1' = N'\,\tau_2' \Rightarrow \exists W.\,N'\,\tau_i' = (W\text{o}N)\,\tau_i'$

That is of finding a nonexpansive and sort preserving unifier of τ_1' and τ_2' which is most-general in the usual sense. The order-sorted discipline of Section 2.2 is easily embedded in the standard unification algorithms, without affecting their correctness. Plain variables may be substituted by any type. Unifying two tagged variables, say $\Delta^{\bar{\tau}}$ and $\Theta^{\bar{\upsilon}}$, consists of substituting a third variable $\Theta^{\bar{\omega}}$ for both (asserted not to occur yet in the unification context nor in any expansion), with ω determined as the most general type which is an instance of both τ and υ; type ω is naturally obtained from the unification of τ and υ. If ω is reduced to a monotype, we should substitute both $\Delta^{\bar{\tau}}$ and $\Theta^{\bar{\upsilon}}$ by the tagged constant ∇^{ω} instead. Unifying a tagged variable $\Delta^{\bar{\tau}}$ with a tagged constant ∇^{μ} is substituting it by the constant, if μ can be unified with τ . ■

This rudimentary algorithm suffices to prove existence of ~-unification algorithms. Another algorithm is suggested in Appendix, convenient for practical purposes. It integrates the expansion and normalization steps and relies on a weaker sufficient condition than that used above to fulfill condition (iii).

As an example, applied to the pair:

$$(\{p{:}\psi\}\{r{:}b\to\alpha\}\Delta^{\forall\alpha.a\to\alpha},\ \{q{:}\delta\}\{r{:}\beta\}\Theta^{\forall\beta\gamma\upsilon.\beta\to\gamma\to\upsilon})$$

Expansion and normalization produces the pair:

$$(\{p{:}\psi\}\{q{:}a\to\alpha_q\}\{r{:}b\to\alpha\}\Delta^{\forall\alpha.a\to\alpha},\ \{p{:}\beta_p\to\gamma_p\to\upsilon_p\}\{q{:}\delta\}\{r{:}\beta\}\Theta^{\forall\beta\gamma\upsilon.\beta\to\gamma\to\upsilon})$$

And unification returns (after normalization):

$$\{p{:}\beta_p\to\gamma_p\to\upsilon_p\}\{q{:}a\to\alpha_q\}\{r{:}b\to\alpha\}\Omega^{\forall\gamma\upsilon.a\to\gamma\to\upsilon}$$

The variables introduced by expansions are arbitrary, but must be different from those in use in the types and those introduced by unification of tagged variables.

Similarly, from the pair: $(\Delta \to \Theta \to \{p{:}\ a\}\Delta,\ \Delta \to \Theta \to \{p{:}\ a\}\Theta)$

Unification produces: $\{p{:}\ \delta\}\Omega \to \{p{:}\ \theta\}\Omega \to \{p{:}\ a\}\Omega$

Note in this last example that the naive ~-unifier $[\Omega/\Delta,\Omega/\Theta]$ is not a ~-mgu. It leads to type $\Omega \to \Omega \to \{p{:}\ a\}\Omega$, $\{\Omega\}$-expansion-equivalent to $\{p{:}\ \delta\}\Omega \to \{p{:}\ \delta\}\Omega \to \{p{:}\ a\}\Omega$, which is clearly less general by $\lesssim$ than the result of unification.

2.5 Further issues and Related work

Tagged types were compared to row types in the introductory section. In the general allowing higher-order sorts (not discussed here), tagged Types and the generalized record types in [14] have different expressiveness since record types do nor admit higher order expansion patterns while tagged types do not allow type variables to be shared between expansion patterns as in the record type $\{\alpha\} \rightarrow \{\beta\} \rightarrow \{\alpha \rightarrow \beta\}$. However, when neither of these features are used, one can easily pass from one of the formalisms to the other.

The usefulness of higher order sorts is questionable as long as no applications taking advantage of them are found. But nontrivial sorts are undoubtedly useful. They serve to encode inheritance in type systems for records, and some of the type systems discussed in Section 3 take advantage of them as well.

As a summary, well-formedness of tagged types reduces to a simple grammatical constraint, tagged variables have a meaning independent of any context, extensibility is simply explained, tagged types can be unified with simple extensions of the standard algorithms, and closed type schemes admit a variety of canonical forms. We believe that tagged types provide a more intuitive and technically simpler alternative to the existing treatments.

3. Typing CCS and Higher Order variants

3.1 The Calculus of Communicating Systems

Given a language of message expressions e and a countable set of labels Σ, our starting process calculus is CCS [9], extended by local declarations (x is a message variable, X is a process variable, a and b are labels in Σ):

$$P ::= a(x).P \mid \bar{b}e.P \mid \tau.P \mid \Sigma_{i \in I} P_i \mid P \mid P' \mid P\backslash a \mid P[b/a] \mid \textbf{if } e \textbf{ then } P \textbf{ else } P' \mid \textbf{letrec } X = P \textbf{ in } Q \mid X \mid \textbf{let } X = P \textbf{ in } Q$$

Message expressions e should include two constants encoding a boolean type. CCS expressions denote processes (or behaviors). They are built, possibly recursively, from the empty summation (written **0**), action prefixing, restriction, renaming and compositions. Actions include proposing a message on a label ($\bar{a}$e.P), accepting a message on a label (a(x).P), and performing an internal move (τ.P).

Processes communicate by rendez-vous. Restrictions and relabelling delimit the scope of label instances. P\a delimits the scope of the instances of a in P to P. P [a/b] makes actions using labels a and b in P appear to the enclosing context as all using label a (but not within P).

A simple rule ensures type-safe communications between processes: Whenever their scopes intersect, two label instances must transmit messages of the same type. The rule is sufficient though not necessary; it is only necessary for the pairs which may actually be involved in a communication, but this cannot be generally inferred at compile-time (unless severely restricting message expressions).

3.2 Typing CCS expressions

Behavior expressions will be assigned *behavior types*, which associate a type with every possible communication label. Behavior types are conveniently defined as a particular class of tagged types. Two sorts are required: a sort $\mathcal{M}$ of *message* types, and a *process* sort $\mathcal{P}$ that can be defined as the tagged sort $\mathcal{T}(\forall\alpha^{\mathcal{M}}.\alpha^{\mathcal{M}})$. Since CCS does not allow process passing (classes e and P above are disjoint), sorts $\mathcal{M}$ and $\mathcal{P}$ are order-unrelated; process types should not be substituted for message type variables, nor message types for process type variables.

CCS behavior types obey the following grammar in which μ (resp. π) ranges over all types of sort $\mathcal{M}$ (resp. $\mathcal{P}$):

μ	::=	$\alpha, \beta, \gamma, \ldots$	message type variables
	|	bool, …	some type constants
π	::=	$\Delta,\ \Theta,\ \ldots$	process type variables
	|	$\{a: \mu\}\pi$	field prefixing

The most general type scheme for a process is $\forall\Delta.\Delta$. The scheme type $\forall-.\ \{a: \mu\}\pi$ (assumed closed) is the type of processes that may communicate massages of type μ on label a. For other labels than a, the constraints are given by type π.

$$\frac{-}{A \vdash x: \mu}\ A(x) \geq \mu \qquad \frac{-}{A \vdash X: \pi}\ A(X) \geq \pi \qquad \frac{A \vdash P: \pi}{A \vdash P: \pi'}\ \pi \sim \pi'\ (\mathit{equal})$$

$$\frac{A \vdash P: \pi}{A \vdash \tau.P: \pi} \qquad \frac{A\oplus\langle x: \mu\rangle \vdash P: \{a: \mu\}\pi}{A \vdash a(x).P: \{a: \mu\}\pi} \qquad \frac{A \vdash e: \mu \quad A \vdash P: \{a: \mu\}\pi}{A \vdash \overline{a}e.P: \{a: \mu\}\pi}$$

$$\frac{A \vdash P: \{a: \mu\}\pi}{A \vdash P\backslash a: \{a: \mu'\}\pi} \qquad \frac{A \vdash P: \{a: \mu\}\{b: \mu\}\pi}{A \vdash P[a/b]: \{a: \mu\}\{b: \mu'\}\pi} \qquad \frac{A \vdash b{:}\mathrm{bool} \quad A \vdash P{:}\pi \quad A \vdash P'{:}\pi}{A \vdash \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ P': \pi}$$

$$\frac{(\forall i\in I)\ \ A \vdash P_i: \pi}{A \vdash \Sigma_{i\in I}\ P_i: \pi} \qquad \frac{A \vdash P: \pi \qquad A\oplus\langle X: \mathit{Gen}(A,\pi)\rangle \vdash Q: \pi'}{A \vdash \textbf{let}\ X{=}P\ \textbf{in}\ Q: \pi'}$$

$$\frac{A \vdash P: \pi \quad A \vdash P': \pi}{A \vdash P \mid P':\ \pi} \qquad \frac{A\oplus\langle X: \pi\rangle \vdash P: \pi \qquad A\oplus\langle X: \mathit{Gen}(A,\pi)\rangle \vdash Q: \pi'}{A \vdash \textbf{letrec}\ X{=}P\ \textbf{in}\ Q: \pi'}$$

Table 3.1. Type inference rules for CCS + let

The type inference rules for CCS expressions are shown in Table 3.1. A maps message and process identifiers to types. As in [7], *Gen*(A,π) is the type scheme obtained by universally quantifying the variables that are free in π but not in A. Input bound variables here are typed like lambda-bound variables in [4] or [7].

The rule for relabelling says that labels a and b must have same types before relabelling, and that b has any type after relabelling. The processes involved in compositions must all have the same (process) type. The empty summation has any process type; it is assumed that the initial typing environment assigns the type scheme $\forall\Delta.\ \Delta$ to **0**. Recursion is typed as in [4], which explains e.g. that the expression (**letrec** A= (p!1.A)\p **in** A) cannot be assigned the scheme $\forall\Delta.\Delta$, but only the weaker $\forall\Delta.\{p{:}int\}\Delta$, though that process cannot communicate through p). The *equal* rule allows to permute fields or remove hidden fields in the process of inference, but expansions are assumed performed by the generic instance rule.

Soundness of the inference system is proven along the lines of [18] by first defining the computations that yield type errors and then proving that if some type could be inferred for some process, then it cannot yield an error when run (the complete proof will appear in [6]).

W(A,P) = case P of				
X	=>	**if** $X \in$ *Dom*(A) **then** (Id, A(X)) **else** FAIL		
τ.P	=>	**let**	(S_1,π)	= W (A,P)
			S_2	= *Unify* (π, *newtype*($\mathcal{P}$))
		in	$(S_2\, S_1,\ S_2\, \pi)$	
$\bar{a}$e.P	=>	**let**	(S_1,μ)	= W (A,e)
			(S_2,π)	= W (S_1 A,P)
			S_3	= *Unify* (π,{a: $S_2\, \mu$}(*newtype*($\mathcal{P}$)))
		in	$(S_3\, S_2\, S_1,\ S_3\, \pi)$	
a(x).P	=>	**let**	μ	= *newtype*($\mathcal{M}$)
			(S_1,π)	= W (A $\oplus$ $\langle$x: $\mu\rangle$,P)
			S_2	= *Unify* (π,{a: $S_1\, \mu$}(*newtype*($\mathcal{P}$)))
		in	$(S_2\, S_1,\ S_2\, \pi)$	
P \ a	=>	**let**	(S_1,π)	= W (A,P)
			S_2	= *Unify* (*newtype*($\mathcal{P}$),π)
		in	($S_2\, S_1$, {a: *newtype*($\mathcal{M}$)}($S_2\, \pi$))	
P [a/b]	=>	**let**	(S_1,π)	= W (A,P)
			μ	= *newtype*($\mathcal{M}$)
			S_2	= *Unify* (π,{a: μ}{b: μ}(*newtype*($\mathcal{P}$)))
		in	($S_2\, S_1$, {b: *newtype*($\mathcal{M}$)}($S_2\, \pi$))	etc,

Table 3.2. Type assignment algorithm for CCS (fragments)

As in [4, 8], type synthesis is reduced to unification. The essential part of the type assignment algorithm is shown in Table 3.2. A maps identifiers to types; Id is the identity substitution. The type assignment algorithm does not rely on generic instantiation to perform the necessary expansions; these are performed by the unification algorithm for tagged types instead (function *Unify*). Function *newtype*(*s*) returns a new type variable of sort *s*.

3.3 Behavior Passing

CHOCS [16] or LCS [3] allow process passing, while retaining the semantics of the restriction operator of CCS. To handle this, the only change required to our previous language is to merge the syntactic classes P and e. In addition, we must allow message type variables to be substituted by process types (i.e. the process sort $\mathcal{P}$ becomes included in the message sort $\mathcal{M}$). The type inference rules are as in Table 3.1, with variables X and x assumed in the same syntactic class. These rules enforces the fact that, whenever an expression is used as a behavior expression, it must have a behavior type. The algorithm in Table 3.2 is easily updated to enforce these constraints (expressions like e.g. p!2.4 would be rejected).

Processes like e.g. (a(x).x) cannot be assigned types in that system (the solution τ should solve $\tau=\{a\colon \tau\}\Delta$). This problem is equivalent to that of typing (**letrec** f = λx.f **in** f) in [4], or that of typing channels communicating themselves in [19]. Clearly, the problem cannot be solved without some form of recursive types, either part of the language of types (we did not investigate the issue for tagged types), or provided at the level of declaration of new type operators (as in ML).

3.4 Parametric Channels

Following a proposal of [1], LCS provides parameterized channels to implement a weak form of delegation. LCS communication ports are constituted of a label, as in CCS, and of an extension (a value admitting equality). Extensions may not appear in restrictions or relabellings. The typing problem with these parameterized channels is essentially the same than for CCS, except that when the scope of two ports intersect, they must have the same extension types too.

To handle these ports, the behavior types of CCS must be enriched with a sort $\mathcal{Q}$ of extension types (possibly a subsort of $\mathcal{M}$). Process types assign to each label a pair of types $\theta\bullet\mu$, with θ of sort $\mathcal{Q}$ and μ of sort $\mathcal{M}$; the process sort $\mathcal{P}$ is redefined as $\mathcal{T}(\forall\alpha^{\mathcal{Q}}\beta^{\mathcal{M}}.\alpha^{\mathcal{Q}}\bullet\beta^{\mathcal{M}})$. Except for this, the treatment is similar to that forCCS.

3.5 Lambda expressions

Adding abstraction and application to our language would permit to define new process combinators. A pipe combinator, connecting a process sending messages on port `out` to another receiving messages on port `inp` could be defined by:

`pipe` = λa . λb. (a \tmp [tmp/out] | b \tmp [tmp/inp]) \tmp

These constructions would be typed as in [7], and the necessary inference rules just added to those in Table 3.1. Extending this way the previous process passing variant of CCS would just require to add function types to the message sort $\mathcal{M}$. The principal type scheme inferred for function `pipe` would be:

$$\forall\alpha\beta\gamma\Delta.\ \{\text{out: }\alpha\}\Delta \rightarrow \{\text{inp: }\alpha,\ \text{tmp: }\beta\}\Delta \rightarrow \{\text{tmp: }\gamma\}\Delta$$

4. Conclusion

Tagged types clearly build upon the work of Wand on row types and that of Rémy on record types. The connections with these formalisms were explained in Section 2 and the Introduction. For practical purposes, tagged types combine, we believe, the intuitiveness of row types with the expressiveness of record types. Further, they require a lighter theoretical treatment.

All these formalisms widely extend the class of expressions of programming languages for which types can be mechanically inferred. For process calculi, our results strictly strengthen the "sorting" methods discussed in the literature and permit type inference for parallel programs based on the CCS paradigm. Compared to the types used in [11], ours do not attempt to represent "polarity" of labels (i.e. to distinguish between labels used for input, output, or both), but it fully handles "label" polymorphism. Compared to the method of [19] (for different calculi), ours has the advantage of assigning nontrivial types to process variables, which permit type reconstruction for processes from the types of their constituent processes. This can be considered an advantage in implementations of declarative languages based upon these paradigms.

Tagged types, and the typing methods presented, have been experimented for several years. The typechecker of the implementations of the language LCS is directly based on the unification algorithm found in Appendix, and on the type inference systems and type assignment algorithms discussed in Section 3.

References

1. E. Astesiano, E. Zucca, "Parametric channels via label expressions in CCS*", Theoretical Computer Science, Vol 33, 1984.
2. B. Berthomieu, "Tagged types - a theory of order sorted types for tagged expressions", Technical Report 93083, LAAS-CNRS, March 1993.
3. B. Berthomieu, T. Le Sergent, "Programming with behaviors in an ML framework, the syntax and semantics of LCS", In Programming Languages and Systems - ESOP'94, Edinburgh, April 1994, LNCS Vol. 788, Springer Verlag, 1994.
4. L. Damas and R. Milner, "Principal type-schemes for functional programs", ACM Symposium on Principles of Programming Languages, 1982.

5. A. Giacalone, P. Mishra, S. Prasad, “Facile: A symmetric integration of concurrent and functional programming”. Int. Journal of Parallel Programming, 18(2), April 1989.
6. C. le Moniès de Sagazan, Typage Polymorphe des Calculs de Processus à Liaisons Noms-Canaux Dynamiques, Ph. D. Thesis, Univ. of Toulouse, to appear.
7. X. Leroy, Typage Polymorphe d’un Langage Algorithmique, PhD Thesis, Univ. of Paris VII, June 1992.
8. R. Milner. “A Theory of Type Polymorphism in Programming Languages”, Journal of Computer and System Science 17, 1978.
9. R. Milner, Communication and Concurrency, Prentice Hall, 1989.
10. R. Milner, “The Polyadic π-calculus: a tutorial”. Technical Report ECS-LFCS-91-180, Dept. of Computer Science, Univ. of Edinburgh, Oct. 91.
11. F. Nielson. “The typed λ-calculus with first-class processes”. In Proceedings of PARLE’89, LNCS Vol. 366, Springer-Verlag, 1989.
12. H.R. Nielson, F. Nielson. “Higher-Order Concurrent Programs with Finite Communication Topology”. ACM Symposium on Principles of Programming Languages, 1994.
13. D. Rémy, “Typechecking records and variants in a natural extension of ML”, ACM Symposium on Principles of Programming Languages, 1989.
14. D. Rémy, “Syntactic Theories and the Algebra of Record Terms”, Research Report 1869, INRIA, March 93.
15. J.H. Reppy, "CML: A Higher Order Concurrent Language”, ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Canada, 1991.
16. B. Thomsen. “A calculus of higher-order communicating systems”, ACM Symposium on Principles of Programming Languages, 1989.
17. B. Thomsen, “A second generation calculus for higher order processes”, Acta informatica, 30:1-59, 1993.
18. M. Tofte, “Operational Semantics and Polymorphic Type Inference”, Ph.D. Thesis, Univ. of Edinburgh, May 1988.
19. V. T. Vasconcelos, K. Honda, “Principal typing schemes in a polyadic π-calculus”, In Proceedings of CONCUR’93, August 1993, LNCS Vol. 715, Springer-Verlag, 1993.
20. M. Wand. “Complete Type Inference for Simple Objects”, In Symposium on Logic in Computer Science, Ithaca, New York, june 1987. (A Corrigenda appeared in LICS 88.)
21. M. Wand. “Type Inference for Objects with Instance Variables and Inheritance”, to appear in Theoretical Aspects of Object-Oriented Programming, C. Gunter and J.C. Mitchell, eds, MIT Press. (Also Northeastern Univ. Technical Report NU-CCS-89-2)

APPENDIX -- The unification algorithm

All instances of a tagged variable are assumed to occur with identically written indices; the variables in these indices not occurring elsewhere. This allows us to omit quantifiers in indices. Tagged variable symbols are assumed not overloaded at several sorts.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\tau_1 \textbf{ Unify } \tau_2 = \tau_1 \mathbf{U}_\varnothing \tau_2} \\
\multicolumn{3}{l}{\quad \text{where } \tau_1 \mathbf{U}_L \tau_2 = \text{case } \{\tau_1,\tau_2\} \text{ of}} \\
\{\alpha_1,\alpha_2\} & \Rightarrow \text{ if } \alpha_1=\alpha_2 \text{ then } [\,] \text{ else } [\alpha_1/\alpha_2] & (1) \\
\{\alpha,\tau\} & \Rightarrow \text{ if } \alpha \textbf{ In } \tau \text{ then } \textbf{Fail} \text{ else } [\tau/\alpha] & (2) \\
\{\tau_1\rightarrow\tau_2,\tau_1'\rightarrow\tau_2'\} & \Rightarrow (S\,\tau_2 \mathbf{U}_\varnothing S\,\tau_2') \circ S \text{ where } S = \tau_1 \mathbf{U}_\varnothing \tau_1' & (3) \\
\{\Delta_1^{\overline{\sigma 1}}, \Delta_2^{\overline{\sigma 2}}\} & \Rightarrow \text{ if } \Delta_1^{\overline{\sigma 1}}=\Delta_2^{\overline{\sigma 2}} \text{ then } [\,] \text{ else } U \circ X1 \circ X2 & (4) \\
 & \text{where } U = \text{if } \textbf{Monotype } (\overline{S}\,\overline{\sigma 1}) & \\
 & \qquad \text{then } [\nabla^{(\overline{S}\,\overline{\sigma 1})}/\Delta_2^{\overline{\sigma 2}}] \circ [\nabla^{(\overline{S}\,\overline{\sigma 1})}/\Delta_1^{\overline{\sigma 1}}] & \\
 & \qquad \text{else } [\Delta_1^{(\overline{S}\,\overline{\sigma 1})}/\Delta_2^{\overline{\sigma 2}}] \circ [\Delta_1^{(\overline{S}\,\overline{\sigma 1})}/\Delta_1^{\overline{\sigma 1}}] & \\
 & \text{and } X1 = \textbf{Expand } (L,\Delta_1^{\overline{\sigma 1}}) & \\
 & \text{and } X2 = \textbf{Expand } (L,\Delta_2^{\overline{\sigma 2}}) & \\
 & \text{where } S = \sigma_1 \mathbf{U}_\varnothing \sigma_2 & \\
\{\Delta^{\overline{\sigma}}, \nabla^{\overline{\mu}}\} & \Rightarrow U \circ X & (5) \\
 & \text{where } U = [\nabla^{(\overline{S}\,\overline{\sigma})}/\Delta^{\overline{\sigma}}] \circ [\nabla^{(\overline{S}\,\overline{\sigma 1})}/\Delta_1^{\overline{\sigma 1}}] & \\
 & \text{and } X = \textbf{Expand } (L,\Delta^{\overline{\sigma}}) & \\
 & \text{where } S = \sigma \mathbf{U}_\varnothing \mu & \\
\{\nabla^{\overline{\mu}}, \nabla^{\overline{\mu}'}\} & \Rightarrow [\,] \quad \text{where } S = \mu \mathbf{U}_\varnothing \mu' & (6) \\
\multicolumn{2}{l}{\{\{p:\tau\}\rho_1, \rho_2 \text{ as } \Delta^{\overline{\sigma}}\} \text{ or } \{\{p:\tau\}\rho_1, \rho_2 \text{ as } \{q:\tau'\}\rho\}} & (7) \\
 & \Rightarrow \text{ if } p \in L \text{ then } \rho_1 \mathbf{U}_L \rho_2 & \\
 & \text{else } (S2\,(S1\,\rho_1) \mathbf{U}_{L\cup\{p\}} S2\,(S1\,\rho_2)) \circ S2 \circ S1 & \\
 & \text{where } S2 = S1\,\tau \mathbf{U}_\varnothing \tau' & \\
 & \text{where } (S1,\tau') = \textbf{Extract } (p,\rho_2) & \\
\_ & \Rightarrow \text{ if } \tau_1=\tau_2 \text{ then } [\,] \text{ else } \textbf{Fail} & (8)
\end{array}
$$

Each case matches the parameters of function $\mathbf{U}_L$ with some set pattern, e.g. both sets $\{\beta,\psi\}$ and $\{\psi,\beta\}$, for some type ψ and plain variable β, match $\{\alpha,\tau\}$ in case (2). Algorithm **U** has a third parameter (its subscript) which is the set of labels for which its arguments have been unified so far (when these are tagged types). That set serves to skip hidden fields and to handle misordering of fields (in case (5)). The nonoverloading hypothesis for tagged variables allows to reuse one of the variable symbols in case (4).

Function **In** implements the occurrence check; it does not check occurrences of variables in hidden fields. Function **Expand** $(L,\Delta^{\overline{\sigma}})$ returns a substitution expanding Δ^{σ} for all labels in L; expansions introduce copies of scheme σ in all fields, using fresh type variables. **Extract** (p,ρ) lookups the type associated with label p in ρ, expanding ρ on the fly if it has no such field; it returns that type and a substitution (empty or an expansion).