Inference of Conversation Types for Distributed Multiparty Systems

Luísa Lourenço Luís Caires

CITI e Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal

Abstract

The Conversation Calculus is a model for distributed communication-centric systems based on the notion of conversation, a generalisation of binary sessions for multi-party interactions over a single shared communication channel. As sessions are disciplined by session types, conversations are disciplined by conversation types, an extension of session types for the conversation interaction model, developed in previous work. Given the fairly rich structure of the underlying type structure, it may not be immediately clear from the proposed type system how types may be inferred for a system, given partial annotations. In this paper, we propose a solution to the conversation type inference problem, proving soundness, completeness and decidability of our algorithm.

1 Introduction

In recent years, there has been an increasing interest in the study and analysis of multiparty service-based system. Several process calculi were designed to model and reason about these systems, namely [5] (based on previous work [4]), and [1]. On top of such models, type systems have been proposed for studying the local and global behavioural correctness of participants in a service-based system. Although type inference for session types has been considered in several works [6, 7], when considering conversation types, given the fairly rich structure of the underlying type structure, and the heavy dependence on a behavioural merge relation, it may not be immediately clear how types may be inferred for a system, given partial annotations.

In this work we present a type inference algorithm for a form of conversation types and show decidability, soundness, and completeness results. Our solution uses standard techniques based on constraint solving (unification). The most challenging aspects of our proposal is the formulation of the particular constraint language used and its combination with the type rules, which has benefited from a direct representation of sequential composition at the level of types.

In section 2 we make a brief introduction to the CC followed by a small example using our language. Section 3, presents our type inference algorithm, an example of its execution, and the correctness results we have obtained. We discuss in section 4 how we can accommodate (iso)recursive types in our type inference algorithm and show that our correctness results are preserved. Finally, we outline some concluding remarks and future work in section 5.

2 Conversation Calculus

The Conversation Calculus (CC) was first introduced in [8] and later refined in [1] and consists in an extension of the π-calculus to allow multiparty conversations (interactions between two or more partners) through a conversation access operator $n \blacktriangleleft [P]$ (P is a process in the context of the conversation n) and context-sensitive communication operators, $l^{\mathsf{d}}!(n)$ and $l^{\mathsf{d}}?(n)$ for output and input, respectively, in either the current conversation context ($\downarrow$) or the enclosing conversation context ($\uparrow$). The syntax is presented in

$$\begin{array}{lll}
\mathtt{P,\ Q} & ::= & \mathbf{0} \mid \mathtt{P} \mid \mathtt{Q} \mid (\nu \mathtt{n})\mathtt{P} \mid \mathbf{rec}\,\mathtt{X.P} \\
 & & \mid \mathtt{X} \mid \mathtt{n} \blacktriangleleft [\mathtt{P}] \mid \sum_{i \in I} \alpha_i . \mathtt{P}_i \\
\mathtt{d} & ::= & \uparrow \mid \downarrow \\
\alpha & ::= & \mathtt{l}^{\mathtt{d}}!(\mathtt{n}) \mid \mathtt{l}^{\mathtt{d}}?(\mathtt{n}) \mid \mathbf{this}(\mathtt{x})
\end{array}$$

Figure 1: Conversation Calculus Syntax

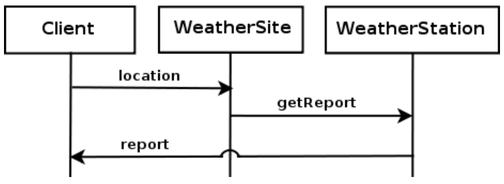

Figure 2: Weather Forecast Message Sequence Chart

Figure 1. We have been developing a concrete distributed language based on CC, which has motivated this work, in this paper we will use sometimes the syntax of our language rather than the formal calculus.

We illustrate our language's syntax through a simple services' use case scenario: a weather forecast service and its client. Upon invocation, the service awaits for the client's location through label location. Then, from the received location, it will ask the nearest weather station to join the on-going conversation (established by the client when invoking the service via **invoke**) and request the desired weather report. The weather station service will, in turn, generate a weather report and send it directly to the client via label report. Notice that the weather station service is capable to communicate directly to the client because it was invoked through the **join** primitive by one of the participants of the conversation, thus is able to join the invoker's conversation instead of creating a new conversation with him. So we have a conversation involving three participants in which one of them dynamically joins. Figure 2 describes the message sequence of our example while Figure 3 shows the code on each participant's site. The message sequence gives a global view of the protocol that every conversation generated by invoking this particular service must comply to (a choreography). Thus, each participant of the conversation must comply with its part of the protocol. Conversation types have been introduced with the aim of statically enforcing correctness of global protocol compliance, given types describing the behaviour of the several participants and of the whole system, for example, $n : [s](B)$ states that site n has a service s whose behaviour is described by behavioural type B.

3 Type Inference

In this section, we will present our type inference algorithm for conversation types and the results obtained, namely we show that it is sound, complete and decidable. Our conversation type system, based on the system of [1], uses judgments of the form $\Gamma \vdash P : T$ where Γ is a set of type declarations, P is the program to be typed and T is a type. In general Γ contains types for remote services, declared in program P using the **remoteType** primitive, and a declaration of the form $this : B$ that describes the current conversation's behaviour B. We show the typing rules for the communication centric fragment of our language in Figure 4. We briefly explain some of the key typing rules. In rule (INVOKE), to typecheck a service invocation we must verify if the body of the invocation has a dual behaviour with the invoked service's behaviour. Then the service invocation is well-typed under the conversation that has the invocation's upper behaviour localised, $loc(\uparrow B)$, i.e. all the message types in the invocation's behaviour that have a up direction correspond to the behaviour of a conversation that invokes the service. In rule (SEND),

```
remoteType WeatherStation: [weatherReport](getReport?(String);report!(String))
site WeatherSite {
  def forecastWeather as {
    val loc = receive(location);
    join weatherReport in
      http://localhost:8000/WeatherStation as { send(getReport); }
  }
};;

site WeatherStation {
  def weatherReport as {
    receive(getReport);
    send(report, generatedReport);
  }
};;

invoke forecastWeather in http://localhost:8000/WeatherSite as {
  send(location, my_location);
  val my_weather_report = receive(report);
  println my_weather_report
};;
```

Figure 3: Code for WeatherSite, WeatherStation, and Client.

we say a send typechecks under the conversation with message type $l!(\beta)$ if the value sent has type β.

The type inference algorithm takes as input a program P, a set of remote types declarations (in a typing environment Γ), an initially empty set of constraints on types R, and an initially empty set of apartness restrictions A. The algorithm outputs the type of program P, the typing environment Γ' where we can typecheck program P, and a set of constraints R' that respects the set of apartness restrictions A'

$$\textbf{typecheck}(\text{P}, \Gamma, \text{R}, \text{A}) = (\text{T}, \Gamma', \text{R}', \text{A}')$$

The algorithm consists of the following steps. First, it transverses the abstract syntax tree, applying typing rules backward if possible. Whenever a type needs to be inferred, a constraint is generated and added to the set R. Finally, the system of equations, represented by all constraints of R is manipulated by applying transformation rules until the system is either in solved form, or type inference fails. During constraint solving, matching labels may need to be synchronised (for e.g., when we have a merge constraint on two dual labels). To ensure linearised usage, we have introduced a new kind of constraint (checked on each transformation step) to state that a label cannot occur in a given type. We denote this as an apartness restriction $l\#B$ (added to the apartness set A), with l being the label that can not occur in type B. As expected, if at any moment a step can not be executed the algorithm aborts since the program must be ill-typed.

The unification algorithm receives as input a constraint set R whose constraints represent a system of equations, and a set of apartness restrictions A. After solving all the constraints in R, the algorithm outputs a set of constraints R' (in particular, a substitution) respecting the set of apartness restrictions A'

$$\textbf{solve}(\text{R}, \text{A}) = (\text{R}', \text{A}')$$

The constraints generated by the type inference algorithm have the form $< E, E' >$ according to the syntax presented in Figure 5b. We have standard constraints like $< x, T >$, stating that a type variable x

$$\frac{\Gamma,\ \mathit{this} : B_1 \vdash P_1 \quad \dots \quad \Gamma,\ \mathit{this} : B_n \vdash P_n}{\Gamma,\ \mathit{this} : B_1 \bowtie \dots \bowtie B_n \vdash \mathrm{P}_1 || \dots || \mathrm{P}_n}\ \text{(PAR)} \qquad \frac{\Gamma,\ \mathit{this} : B_1 \vdash P_1 \quad \Gamma,\ \mathit{this} : B_2 \vdash P_2}{\Gamma,\ \mathit{this} : B_1 ; B_2 \vdash \mathrm{P}_1;\mathrm{P}_2}\ \text{(SEQ)}$$

$$\frac{\Gamma,\ \mathit{this} : B,\ n : [s](B_1) \vdash P}{\Gamma,\ \mathit{this} : B_1 \bowtie B,\ n : [s](B_1) \vdash \textbf{join}\ \mathrm{s}\ \textbf{in}\ \mathrm{n}\ \textbf{as}\ \{\ \mathrm{P}\ \}}\ \text{(JOIN)} \qquad \frac{}{\Gamma,\ \mathit{this} : l?(\beta) \vdash \textbf{receive}(\mathrm{l}){:}\beta}\ \text{(RECV)}$$

$$\frac{\Gamma,\ \mathit{this} : B \vdash P}{\Gamma,\ \mathit{this} : [s](\downarrow B);\mathit{loc}(\uparrow B) \vdash \textbf{def}\ \mathrm{s}\ \textbf{as}\ \{\ \mathrm{P}\ \}}\ \text{(DEF)} \qquad \frac{\Gamma \vdash E : \beta}{\Gamma,\ \mathit{this} : l!(\beta) \vdash \textbf{send}(\mathrm{l},\mathrm{E})}\ \text{(SEND)}$$

$$\frac{\Gamma,\ \mathit{this} : B,\ n : [s](B_1) \vdash P \quad B_1 = \overline{\downarrow B}}{\Gamma,\ \mathit{this} : \mathit{loc}(\uparrow B),\ n : [s](B_1) \vdash \textbf{invoke}\ \mathrm{s}\ \textbf{in}\ \mathrm{n}\ \textbf{as}\ \{\ \mathrm{P}\ \}}\ \text{(INVOKE)} \qquad \frac{\Gamma,\ \mathit{this} : B \vdash P}{\Gamma,\ n : B \vdash \textbf{site}\ \mathrm{n}\ \{\mathrm{P}\}}\ \text{(SITE)}$$

$$\frac{\Gamma,\ \mathit{this} : B_1 \vdash E : \beta \quad \Gamma,\ \mathit{this} : B_2,\ x : \beta \vdash P}{\Gamma,\ \mathit{this} : B_1 ; B_2 \vdash \textbf{let}\ \mathrm{x} = \mathrm{E}\ \textbf{in}\ \{\ \mathrm{P}\ \}}\ \text{(LET)}$$

$$\frac{\Gamma,\ \mathit{this} : B_1 \vdash P_1 \quad \dots \quad \Gamma,\ \mathit{this} : B_n \vdash P_n}{\Gamma,\ \mathit{this} : \&\{l_1 : B_1;\ \dots;\ l_n : B_n\} \vdash \textbf{select}\ \{\ \mathrm{l}_1{:}\ \mathrm{P}_1;\ \dots;\ \mathrm{l}_n{:}\ \mathrm{P}_n\}}\ \text{(SELECT)}$$

$$\frac{\begin{array}{c} \mathit{if}\ l_i = l_j\ \mathit{then}\ B_i = B_j \quad i,j \in \{1,\ \dots,\ n\} \quad m <= n \\ \Gamma \vdash E_1 : \mathit{Bool} \quad \dots \quad \Gamma \vdash E_n : \mathit{Bool} \quad \Gamma,\ \mathit{this} : B_1 \vdash P_1 \quad \dots \quad \Gamma,\ \mathit{this} : B_n \vdash P_n \quad \Gamma,\ \mathit{this} : B_d \vdash P_d \end{array}}{\Gamma,\ \mathit{this} : \oplus\{l_1 : B_1;\ \dots;\ l_m : B_m\} \vdash \textbf{switch}\ \{\ \textbf{case}\ (\mathrm{E}_1)\textbf{do}\ \mathrm{l}_1{:}\ \mathrm{P}_1;\dots;\ \textbf{case}\ (\mathrm{E}_n)\textbf{do}\ \mathrm{l}_n{:}\ \mathrm{P}_n;\ \textbf{default do}\ \mathrm{l}_i{:}\ \mathrm{P}_d\}}\ \text{(SWITCH)}$$

Figure 4: Some Typing Rules of our Type System.

has type T (either behavioural type B or a basic type β). In the unification algorithm, these are treated using the standard transformation rules for variable elimination and type equality [3], a solvable system terminates in a system in solved form, that corresponds to a substitution.

A distinguishing aspect of our constraint structure is a merge constraint on types of the form $< x, \bowtie(B,B') >$, that constrains type variable x to be a "composition" of behavioural types B and B' (this operation is defined by a *merge relation*, see [1]). Merge constraints are necessary to approximate the type of a parallel composition, rule (PAR) in Figure 4, (where synchronisation can happen) or when we invoke a service via the join primitive, rule (JOIN) in Figure 4, (since we merge the behaviours of the invoked service with the client's). Thus we need to be able to represent the merge of all behaviour in the composition such that casual ordering is kept and interleaves are avoided unless there is a synchronisation: this way, the most general (less serialised) behaviour is computed. Merge constraints are solved using a set of transformation rules that represent the merge relation on behavioural types. We now present the transformation rules.

Definition 3.1 (Non Interference)**.** *We say two behavioural types, B_i and B_j, do not interfere with each other, denoted as $B_i \# B_j$, if B_i has no label that can synchronise with some label in B_j, and conversely.*

$$
\begin{array}{lcl}
C & ::= & [s](B) \\
B & ::= & B_1 \,|\, B_2 \;|\; \mathbf{0} \;|\; B_1;B_2 \;|\; M \\
 & & |\; rec\, X.B \;|\; X \\
 & & |\; \oplus\{l_1 : B_1;\; \ldots;\; l_n : B_n\} \\
 & & |\; \&\{l_1 : B_1;\; \ldots;\; l_n : B_n\} \\
M & ::= & lp^d(\beta) \\
\beta & ::= & Int \;|\; Bool \;|\; String \;|\; Array(\beta) \\
 & & |\; \beta \xrightarrow{B} \beta' \;|\; Ref(\beta) \;|\; Unit \\
p & ::= & ! \;|\; ? \;|\; \tau \\
d & ::= & \uparrow \;|\; \downarrow
\end{array}
$$

(a) Syntax of Types

$$E \;::=\; B \,|\, \beta \,|\, \bowtie(E,E') \,|\, x$$

(b) Syntax of constraints on types

Figure 5: Syntax of Types and Constraints

Definition 3.2 (Transformations Rules)**.** *Let R denote a system of equations, t a term, A an apartness constraint set, and T a type. We define the transformations rules, $R \Longrightarrow^A R'$ (if R does not violate any apartness constraint in A, then we can transform to a system R' that also complies with A), as follows:*

Trivial:

$$\{< t,t' >\} \cup R \Longrightarrow^A_{triv} R$$

where $t \equiv t'$

Variable Elimination:

$$\{< x,T >\} \cup R \Longrightarrow^A_{elem} \{< x,T >\} \cup R[{}^{T}/_{x}]$$

such that $x \notin Var(T)$

Merge Trivial:

$$\{< x, \bowtie(B) >\} \cup R \Longrightarrow^A_{merge\_trivial} \{< x,B >\} \cup R$$

Merge Inact:

$$\{< x, \bowtie(B_1,\ldots,B_{i-1},0,B_{i+1},\ldots,B_n) >\} \cup R \Longrightarrow^A_{merge\_inact}$$
$$\{< x, \bowtie(B_1,\ldots,B_{i-1},B_{i+1},\ldots,B_n) >\} \cup R$$

Merge Parallel:

$$\{< x, \bowtie(B,\ldots,B_1|B_2,\ldots,B') >\} \cup R \Longrightarrow^A_{merge\_par}$$
$$\{< x, \bowtie(B,\ldots,B_1,B_2,\ldots,B) >\} \cup R$$

Merge Sync:

$$\{< x, \bowtie(B_1,\ldots,B_i;lp_1^d;B_i',\ldots,\; B_j;lp_2^d;B_j',\ldots,B_n) >\} \cup R \Longrightarrow^A_{merge\_sync}$$

$$\{<x,(B_i \mid B_j);l\tau^d;y>\} \cup \{<y,\bowtie(B_1,\ldots,B_i',\ldots,B_j',\ldots,B_n)>\} \cup \sigma(R)$$

where p_1 is the opposite polarity of p_2, $A = A \cup \{l\#y\}$, and $B_i\#B_j$ and for all B_k, $B_i\#B_k$ and $B_j\#B_k$ with $k \in \{1,\ldots,n\}$ and $k \neq i \neq j$, and $\sigma = [^{(B_i \mid B_j);l\tau^d;y}/_x]$.

Merge Choice Sync:

$$\{<x,\bowtie(B_1,\ldots,B_i;C;B_i',\ldots,\ B_j;D;B_j',\ldots,B_n)>\} \cup R \Longrightarrow^{A}_{merge\_sync2}$$
$$\{<x,(B_i \mid B_j);\oplus\{l_1:y_1,\ldots,l_n:y_n\};y>\}\cup$$
$$\{<y,\bowtie(B_1,\ldots,B_i',\ldots,B_j',\ldots,B_n)>\}\cup$$
$$\{<y_1,\bowtie(B_{c1},B_{c1}')>\} \cup \ldots \cup \{<y_n,\bowtie(B_{cn},B_{cn}')>\} \cup \sigma(R)$$

where $C = \&\{l_1:B_{c1},\ \ldots,\ l_n:B_{cn}\}$ and $D = \oplus\{l_1:B_{c1}',\ \ldots,\ l_n:B_{cn}'\}$, and $B_i\#B_j$ and for all B_k, $B_i\#B_k$ and $B_j\#B_k$ with $k \in \{1,\ldots,n\}$ and $k \neq i \neq j$, and $\sigma = [^{(B_i \mid B_j);\oplus\{l_1:y_1,\ldots,l_n:y_n\};y}/_x]$.

We will now explain one of the rules that represent our merge relation, namely the Merge Sync rule. This rule is applied when a synchronisation is possible between the types the terms represent. Its application ensures that when we synchronise a label then the preceding type on each side, B_i and B_j, must not interfere with each other, $B_i\#B_j$ and thus can be safely composed into a parallel composition of types, $B_i|B_j$. A new equation is generated to merge the remaining types of the merge along with the remainder of both sides' types, B_i' and B_j'. Since labels must be used linearly, we impose an apartness restriction, $l\#y$, stating that the synchronised label, l, can not occur in the new equation, y. Lastly, we eliminate the solved variable x in the remaining constraints in R by applying the substitution σ to R.

The unification algorithm is confluent. This implies that our type inference algorithm is deterministic since the application of typing rules is syntax driven, and also due to the non-interference conditions on the merge relation mentioned above. The solution returned is represented as a mapping on variables to types, i.e. as a substitution for the variables of the system. The type inference algorithm then succeeds if such a solution exists. Furthermore, our algorithm always determines the most general type since we serialise the composition of two types only when there is a synchronisation between them, thus keeping them as general as possible by composing into a parallel composition of types.

We illustrate the application of our algorithm to the code of the weather forecast service's site in Figure 3.

As input we have

P = code of weather forecast's site as shown in Figure 3
Γ = { WeatherStation:[weatherReport](getReport?(String);report!(String)) }
R = A = ∅

So **typecheck**(P, Γ, R, A) takes the following steps:

1. Inductively, applies type inference rules;
2. When typechecking the join primitive, a type variable **x** is created as well as a constraint to represent the merge of the primitive's body's behaviour with the behaviour of the invoked service, <x, ⋈(getReport!(String), getReport?(String);report!(String))> that is added to R;
3. The constraint is solved upon the typecheck of the site primitive and the algorithm terminates.

The algorithm outputs (Γ ∪ WeatherSite:[*weatherForecast*](location?(z);B), R, A′) where B is the type obtained by the unification algorithm (function **solve**) when solving the constraint on **x**, and A' the resulting apartness set.

In step 3 the unification algorithm is called with the following input:

R = { <x, ⋈(getReport!(String), getReport?(String);report!(String))> }
A = ∅

So **solve**(R, A) takes the following steps:

<x, ⋈(getReport!(String), getReport?(String);report!(String))> $\Longrightarrow^{A}_{merge\_sync}$

<x, getReportτ(String);y> ∪ <y, ⋈(∅, report!(String))> $\Longrightarrow^{A'}_{merge\_inact}$
with A' = { getReport#y}

<x, getReportτ(String);y> ∪ <y, ⋈(report!(String))> $\Longrightarrow^{A'}_{merge\_trivial}$

<x, getReportτ(String);y> ∪ <y, report!(String)>

We then obtain B = getReportτ(String);report!(String), R = { <x, getReportτ(String);y>, <y, re-port!(String)> }, and A′ = { getReport#y }.

For a negative case, suppose that we make use of label **getReport** instead of the label **report** to transmit the requested report. This would violate the condition of labels being used linearly because then, at one point, we would have two receivers for the empty message sent by the **forecastWeather** service. This type of errors are detected by the algorithm when solving the constraints. In our example, the unification algorithm would instead solve variable **y** with type **getReport!(String)** which would violate the apartness restriction **getReport#y** and therefore the unification algorithm would abort. Thus typechecked programs always comply with linear usage of labels inside conversations.

We denote a substitution application as the application of a constraint set to a typing environment, R(Γ), and define it as being the substitution of all occurrences of a type variable in the input environment with its corresponding type if, and only if, the constraint associated with the type variable is solved in R.

We conclude by presenting the main results for the type inference algorithm which are the decidability, soundness and completeness of the algorithm. The first states that if a program *P* can be typechecked by the typechecking algorithm, then there is a type derivation for any type instance of the generated inferred type.

Theorem 3.3 (Soundness of Typechecking Algorithm)**.** *Let P be a program,* Γ, Γ' *type environments , T a type, A a set of apartness constraints, and R a constraint set.*

Assume $\mathtt{typecheck}(\mathtt{P},\Gamma,\emptyset,\emptyset) = (\mathtt{T},\Gamma',\mathtt{R},\ \mathtt{A})$. *Then* $\mathtt{R}(\Gamma') \vdash \mathtt{P} : \mathtt{R}(\mathtt{T})$ *and there is a substitution* θ *such that* $\theta(\mathtt{R}(\Gamma')) \vdash \mathtt{P} : \theta(\mathtt{R}(\mathtt{T}))$

The second results states that no typing is lost; if a program *P* is typable then the type-checking algorithm terminates with a typing, of which the given typing is an instance.

Theorem 3.4 (Completeness of Typechecking Algorithm)**.** *Let P be a program,* Γ *a typing context,* Γ' *a typing context containing only type declarations of remote services, and T a type*

Assume $\Gamma \vdash \mathtt{P} : \mathtt{T}$. *Then* $\mathtt{typecheck}(\mathtt{P},\Gamma',\emptyset,\emptyset) = (\mathtt{T}',\Gamma'',\mathtt{R}',\mathtt{A}')$ *and there is a substitution* θ *such that* $\theta(\mathtt{R}'(\mathtt{T}')) = \mathtt{T}$ *and* $\theta(\mathtt{R}'(\Gamma'')) = \Gamma$ *and* $\theta(\mathtt{R}'(\Gamma')) \subseteq \Gamma$

Decidability follows from the termination proof, which depends essentially on the termination of the unification algorithm, in turn based on standard well-founded orderings. In particular, we define a pair $< m,n >$ such that n is the number of variables unsolved in R and m the sum of the sizes of each term in R, then the lexicographic order of such pairs is a well-founded relation. We then prove that every

$$\frac{\Gamma,\ this: B; X \vdash P}{\Gamma,\ this: rec\ X.B; X \vdash \textbf{rec X}.\mathrm{P}}$$

(a) Rec rule

$$\frac{}{\Gamma,\ this: X \vdash X}$$

(b) Var rule

Figure 6: Typing Rules for Recursive Behaviour Constructions.

transformation sequence terminates since each transformation results in a system where the pair $< m,n >$ is smaller under the lexicographic ordering.

4 Recursive Types

We have mainly focused on the finite part of conversation types, in this section we discuss how a simple system of iso-recursive is to be accommodated using standard techniques for unifying recursive equations on our constraint's language. Although we expect that general equi-recursive types may be also accommodated along the lines of [2], when dealing with recursive definitions in our language (such as a recursive functions), we don't focus on that issue in this paper. Instead, we develop here a simple solution, based on the simple interpretation of recursive types. Regarding recursive behaviour's constructions like CC's **rec X.P** (Figure 6), these would not introduce a recursive equation and therefore their treatment in our theory consists in adding two transformation rules to merge two recursive behavioural types, and to merge two recursive variables, respectively:

$$\{< x, \bowtie(B_1,\dots,B_i; rec\ X.B; B_i',\dots,\ B_j; rec\ X.B'; B_j',\dots,B_n) >\} \cup R \Longrightarrow^{A}_{merge\_rec}$$
$$\{< x, (B_i \mid B_j); rec\ X.y >\} \cup \{< y, \bowtie(B_1,\dots,B; B_i',\ \dots,\ B'; B_j',\dots,B_n) >\} \cup \sigma(R)$$

where $B_i \# B_j$ and for all B_k, $B_i \# B_k$ and $B_j \# B_k$ with $k \in \{1,\dots,n\}$ and $k \neq i \neq j$, and $\sigma = \left[{}^{(B_i \mid B_j); rec\ X.y}/_{x}\right]$.

$$\{< x, \bowtie(B_1,\dots,B_i; X; B_i',\dots,\ B_j; X'; B_j',\dots,B_n) >\} \cup R \Longrightarrow^{A}_{merge\_recvar}$$
$$\{< x, (B_i \mid B_j); X; y >\} \cup \{< y, \bowtie(B_1,\dots,B_i',\dots,B_j',\dots,B_n) >\} \cup \sigma(R)$$

where $B_i \# B_j$ and for all B_k, $B_i \# B_k$ and $B_j \# B_k$ with $k \in \{1,\dots,n\}$ and $k \neq i \neq j$, and $\sigma = \left[{}^{(B_i \mid B_j); X; y}/_{x}\right]$.

We illustrate the application of these new rules on our previous example with a minor change, Figure 7. Notice that the **while** construction can be perceived as CC's **rec** X.P construction. In this case, in step 3 of the typechecking procedure, the unification algorithm is called with the following input:

R = { <x, ⋈(rec X.getReport!(String);X, rec X.getReport?(String);X;report!(String))> }
A = ∅

So **solve(R, A)** takes the following steps:

<x, ⋈(rec X.getReport!(String);X, rec X.getReport?(String);X;report!(String))>

$$\Longrightarrow^{A}_{merge\_rec}$$

<x, rec X.y> ∪ <y, ⋈(getReport!(String);X, getReport?(String);X;report!(String))>

```
remoteType WeatherStation: [weatherReport](rec X.getReport?(String);X;report!(String))
site WeatherSite {
  def forecastWeather as {
    val loc = receive(location);
    join weatherReport in
       http://localhost:8000/WeatherStation as {
         while(cond) do
           send(getReport);
       }
  }
};;
```

Figure 7: Weather Forecast Site Code Revisited.

$$\Longrightarrow^{A}_{merge\_sync}$$

<x, rec X.y> ∪ <y, getReportτ(String);z> ∪ <z, ⋈(X, X;report!(String))>

$$\Longrightarrow^{A'}_{merge\_recvar}$$

with A' = { getReport#z}

<x, rec X.y> ∪ <y, getReportτ(String);z> ∪ <z, X;w> ∪ <w, ⋈(∅, report!(String))>

$$\Longrightarrow^{A'}_{merge\_inact}$$

<x, rec X.y> ∪ <y, getReportτ(String);z> ∪ <z, X;w> ∪ <w, ⋈(report!(String))>

$$\Longrightarrow^{A'}_{merge\_trivial}$$

<x, rec X.y> ∪ <y, getReportτ(String);z> ∪ <z,X;w> ∪ <w, report!(String)>

We then obtain B = rec X.getReportτ(String);X;report!(String), A$'$ = { getReport#z }, and R = { <x, rec X.y>, <y, getReportτ(String);z>, <z, X;w>, <w, report!(String)> }.

Our basic results are not affected by our treatment of recursion. Namely, decidability is preserved since the new transformation rules preserves the well-founded ordering defined for the non-recursive case.

5 Concluding Remarks

We have presented a type inference algorithm for conversation types and proved it to be decidable, sound and complete. Our contributions essentially focus on the finite aspects, which are already challenging, due to the parallel-sequential behavioural algebra embedded in conversation types, and the need of coping with the behavioural merge of behaviours originating in multiple interaction partners, including dynamic conversation join and leave. We also showed how to accommodate recursive behaviour's constructions in our type inference algorithm and proved that our results are still preserved. For future work we wish to devise a subtyping algorithm for conversation types, and more general solution for handling recursion.

Acknowledgements. We thank Hugo Vieira and the anonymous referees for his insightful comments and suggestions regarding this paper. This work was supported by a grant from EU Project SENSORIA, and CMU-Portugal INTERFACES, funded by FCT/MCTES and ICTI.

References

[1] Luís Caires and Hugo Torres Vieira. Conversation types. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2009.

[2] Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983.

[3] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general e-unification. *Theor. Comput. Sci.*, 67(2&3):203–260, 1989.

[4] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[5] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284. ACM, 2008.

[6] Leonardo Gaetano Mezzina. How to infer finite session types in a calculus of services and sessions. In Doug Lea anzd Gianluigi Zavattaro, editor, *Coordination Models and Languages, 10th International Conference, COORDINATION 2008. Proceedings*, volume 5052 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2008.

[7] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009.

[8] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus: A model of service-oriented computation. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.