Inference of session types from control flow

Peter Collingbourne[1]

Department of Computing
Imperial College London
London, SW7 2AZ
United Kingdom

Paul H J Kelly[2]

Department of Computing
Imperial College London
London, SW7 2AZ
United Kingdom

Abstract

This is a study of a technique for deriving the session type of a program written in a statically typed imperative language from its control flow. We impose on our unlabelled session type syntax a well-formedness constraint based upon normalisation and explore the effects thereof. We present our inference algorithm declaratively and in a form suitable for implementation, and illustrate it with examples. We then present an implementation of the algorithm using a program analysis and transformation toolkit.

Keywords: Session types, imperative programming, control flow, type inference, program analysis

1 Introduction

The session type [10] is a means of characterising dyadic interaction between processes over a communication channel. A session type is a property of a session, a communication link established over a channel. Process interactions are expressed as a sequence of communication actions, and any communication taking place over the session with which the type is associated must conform to the sequence of actions. Although the roots of session typing can be traced to the π-calculus [15], it has also been applied to a wide range of programming paradigms, including object-oriented imperative programming [6].

A session type may take a number of forms, but let us presently consider a session type consisting of a graph where a communicating process is associated with a single node in a graph. A communication action must be conformant with an outgoing

[1] Email: pcc03@doc.ic.ac.uk
[2] Email: p.kelly@imperial.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

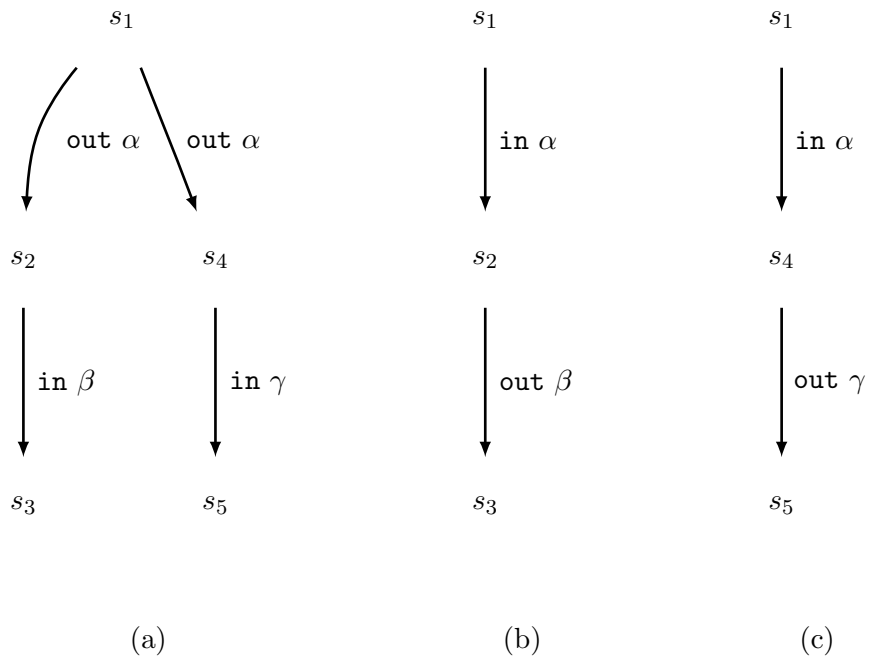

Fig. 1. Three exemplary session type graphs.

arc at the process's current node, and causes the appropriate arc of the session type to be followed, based on the type of the communication action. Figure 1 shows three session type graphs (a), (b) and (c), of which (a) and (b), and (a) and (c) are purportedly compatible with each other (due to the common subgraph). Let us consider two processes A and B with respective session type graphs (a) and (b). Both processes start in state s_1. Firstly process A sends a message of type α and transitions to state s_4. When process B receives this message it transitions to state s_2. Process B then sends a message of type β. However, process A cannot process this message as, according to its session type graph, it may only receive messages of type γ. A similar situation arises with interacting processes A and C of types (a) and (c) where process A first transitions to state s_2 upon receiving the message of type α. We can thus conclude that it is impossible to construct a session type such that processes with that session type may safely communicate with processes with session type (a).

Note that process A makes an internal choice about which of its two branches is taken before sending the value of type α. Notice further that no information was passed from process A to its peer regarding its choice of branch. This is what we expect in a session type system with *implicit choice*. In this paper we shall explore how the above situation may arise in a session type inference system that produces session types with implicit choice and how we may detect it.

We claim the following contributions:

- The first session type inference algorithm known to the authors for statically-typed imperative languages with a session type syntax based on implicit choice;

- A normalisation-based well-formedness constraint for session types with a syntax based on implicit choice;
- A property that ensures session type safety for session types with a syntax based on implicit choice which simultaneously permit both inputs and outputs, known as the safe directionality property;
- An implementation of session type inference, based on a session-based communicating process library for C++.

1.1 Background: implicit choice in session types

In most existing literature, session types are expressed as expressions with a specified syntax. Session type syntax is generally recursive. This allows for arbitrary composition of communication actions in whichever form fits the structure of the program. A session type may take a number of forms, whose semantics we shall briefly describe. A session type may be an *action*. An action specifies a communication direction (in or out) and type (this may be a language-specific primitive type or, in the case of delegation [6], another session type). An action represents, dependent on the communication direction, the reception or transmission of a value (or session) of the specified type. A session type may also be the *sequential composition* of two or more session types. The session type that is the composition of one or more session s_1, $s_2 ... s_n$ represents the actions in s_1, followed sequentially by those in s_2 and so on up to s_n. A session type may also be a *choice* between a number of sessions s_1, $s_2 ... s_n$. The process of making a decision between these choices is described in the following paragraph. A session type may also be the *terminating session type*. A session with the terminating session type may not perform any communication actions or change its type. It may only close the communication channel.

A process may commit to one of these choices either passively or actively, and either implicitly or explicitly. If a process makes the choice actively, then the choice was made by the process based on its choice of communication steps. If the process makes the choice passively, then the choice was made based upon the active choice made by its peer.

Under implicit choice, the process performs a communication action that is consistent with only one of the choices. Note that the process's role in committing to the choice is either active or passive depending on the direction of the communication action. If the process transmits data, its role is active; if it receives data, its role is passive and its choice depends on the type of the data received. [5] is an example of a system which contains a form of implicit choice.

Under explicit choice, the process performs some action other than a communication action that has the effect of selecting a particular choice. The literature includes a number of ways of expressing explicit choice. In [10], choice is represented by the & and $\oplus$ binary operators. A process whose session type is of the form $s_1 \& s_2$ makes a passive choice between s_1 and s_2, whereas a process whose session type is of the form $s_1 \oplus s_2$ makes an active choice via the inl and inr operators. In [8,9], each choice is annotated with a *label*. The process making the active choice transmits the label corresponding to its desired choice, and the process making the passive

choice chooses the session type corresponding to the label it receives.

After performing a communication action, or in the case of explicit choice another relevant action, the current type of the session mutates in order to reflect the current state of the channel. If a choice has been made, the session type is replaced with the type corresponding to the choice that has been made. If a communication action has occurred, the type representing that communication action is removed from the beginning of the current session type. The resultant session type is known as the continuation type of that session type under the given action.

Compatibility [8] is a relation between session types that indicates whether two programs with specified session types are guaranteed to communicate with each other safely; that is, without any possible protocol incompatibilities at runtime. The compatibility relation has in particular been useful in specifying and verifying *contracts* between two parties: by verifying compatibility before a potential communication takes place it is possible to check that no protocol incompatibilities may possibly occur between the two parties – provided that both parties abide by their session type contracts. It is clear that a necessary condition for a session type to be compatible with another is that it must accept at least the data types which the other may emit. We shall see a more formal definition of this concept later.

The goal of this work is to investigate means for inferring a session type using program analysis techniques given an imperative program consisting of a sequence of communication actions. In some process formalisms, such as the π-calculus as described in [10], there is normally no need for an inference algorithm, as the construction rules for a process implicitly perform typing. Here we adopt a language-neutral approach better suited to the structure of imperative programs, using control flow and expression typing information provided by the host language to derive an appropriate session type. In contrast to many other studies of session type inference [10,6], our session types use implicit choice. Our rationale for this design decision is that implicit choice provides a closer mapping between the behaviour of the program and its session type. Additionally, it frees the programmer from the burden of providing a tag name for each communication action in an untyped program. We shall explore the consequences of this decision on our type inference technique.

Our type inference tool allows us to decide interface compatibility between programs without the need for a formal protocol specification beyond that implied by the programs' typing and control flow structures. For example, a programmer can write a server communication program to be used in a client/server architecture and expect any clients with which it communicates to be constrained by its protocol without any extra work. There are two key steps in such a process: firstly, our inference algorithm is employed to determine the session types governing those programs for which we wish to decide compatibility; secondly, compatibility is checked via the host language's type system, a necessary foundation of such compatibility checking being the ability to augment, or simulate the augmentation of, the host language's type system to recognise the session type's subtyping relation.

```
while (1) {
  int x;
  recv_choice (s) {
    case Req1:
      s.receive(Req1(x));
      s.send(x+1);
    case Req2:
      s.receive(Req2(x));
      s.send(x+2);
    case Quit:
      s.receive(Quit());
      s.close();
      return; // exit subroutine
  }
}
```

Fig. 2. Simple pseudocode server process.

```
int x;
s.send(Req1(42));
s.receive(x);
s.send(Quit());
s.close();
```

Fig. 3. Simple pseudocode client process.

1.2 Example

Consider the server program shown in Figure 2, which we wish to interface with the client program shown in Figure 3. We verify by inspection that these two programs will interface with each other correctly, and so does our system by means of session type inference and compatibility checking.

Our system can infer the types of both processes. The inferred type for session s in Figure 2 is

$$\mu t.(\texttt{in}\ \text{Req1}.\texttt{out}\ \text{int}.t|\texttt{in}\ \text{Req2}.\texttt{out}\ \text{int}.t|\texttt{in}\ \text{Quit}.\texttt{end})$$

and the inferred type of session s in Figure 3 is

$$\texttt{out}\ \text{Req1}.\texttt{in}\ \text{int}.\texttt{out}\ \text{Quit}.\texttt{end}$$

Using these types the augmented type system of the host language will verify compatibility.

1.3 Definitions

Our inference system is specified in two distinct ways. Firstly we shall provide a set of inference rules and a methodology for applying them in order to derive a session type. Secondly we shall describe a graph-based implementation technique for the

$$
\begin{array}{rcll}
D & ::= & \texttt{"in"} \mid \texttt{"out"} & \\
ST & ::= & \texttt{"}\mu\texttt{"}\ t\ \texttt{"."}\ ST & \text{(Mu)} \\
 & \mid & \texttt{"end"} & \text{(End)} \\
 & \mid & t & \text{(TV)} \\
 & \mid & \texttt{"("}\ ST\ \texttt{"|"}\ ST\ \texttt{")"} & \text{(Choice)} \\
 & \mid & \texttt{"("}\ ST\ \texttt{"."}\ ST\ \texttt{")"} & \text{(Seq)} \\
 & \mid & D\ VT & \text{(Action)}
\end{array}
$$

Fig. 4. Syntax for a Ninja session type.

algorithm. The graphs used by this technique are based on finite automata [18] and thus we employ a number of techniques from this field, including the subset construction [18].

Ninja is a specification for a component-based imperative language extension. Ninja can be considered an implementation of common component models such as architecture description languages as shall be described in Section 2. It may extend most imperative languages, however our implementation is for the C++ language and is known as Ninja-C++. We describe the implementation of Ninja-C++ and of a type inference tool for it.

Figure 4 shows the syntax for session types in the Ninja language. Note that in informal discussions we use the associativity of "|" and "." to elide parentheses wherever possible. Most of the semantics is clear with reference to Section 1, however note the syntax elements (Mu) and (TV). These are standard [17] syntax elements used for recursive type definitions. (Mu) declares a type variable t of arbitrary name for use. Corresponding (TV) elements are found within the (Mu) element and are equivalent to the whole of the corresponding outer (Mu).

Ninja is a component based language; components are active and are known as *participants*. Participants communicate with each other over channels of specified session types, which means their session types must be compatible. We proceed to introduce our notion of compatibility as initially defined by [8] and extended by, among others, [20]. In order to determine compatibility we must first define equivalence, continuation, subtyping and duality for our session type syntax. Equivalence ($\equiv$) is the smallest relation that satisfies the rules given in Figure 5. The continuation type of a session type under a given communication action may be derived using the rules given in Figure 6.

Many of the equivalence and continuation rules are self explanatory, however we feel it necessary to give a justification of rule (|Dist $\leftarrow$). This will be done in Section 3.1 after the necessary background has been described.

Definition 1.1 Free names over session types.

$$\begin{aligned}
\mathsf{FN}(\mu n.s) &= \mathsf{FN}(s) \setminus n \\
\mathsf{FN}(\mathtt{end}) &= \emptyset \\
\mathsf{FN}(n) &= \{n\}, n \text{ a type variable} \\
\mathsf{FN}((t_1|t_2)) &= \mathsf{FN}(t_1) \cup \mathsf{FN}(t_2) \\
\mathsf{FN}((t_1.t_2)) &= \mathsf{FN}(t_1) \cup \mathsf{FN}(t_2) \\
\mathsf{FN}(a) &= \emptyset
\end{aligned}$$

Definition 1.2 Input and output domains.

$$\begin{aligned}
\mathsf{idom}(\mu n.s) &= \mathsf{idom}(s) & \mathsf{odom}(\mu n.s) &= \mathsf{odom}(s) \\
\mathsf{idom}(\mathtt{end}) &= \emptyset & \mathsf{odom}(\mathtt{end}) &= \emptyset \\
\mathsf{idom}(\mathrm{n}) &= \emptyset, n \text{ a type variable} & \mathsf{odom}(\mathrm{n}) &= \emptyset, n \text{ a type variable} \\
\mathsf{idom}((t_1|t_2)) &= \mathsf{idom}(t_1) \cup \mathsf{idom}(t_2) & \mathsf{odom}((t_1|t_2)) &= \mathsf{odom}(t_1) \cup \mathsf{odom}(t_2) \\
\mathsf{idom}((t_1.t_2)) &= \mathsf{idom}(t_1) & \mathsf{odom}((t_1.t_2)) &= \mathsf{odom}(t_1) \\
\mathsf{idom}(\mathtt{in}\ t) &= \{t\} & \mathsf{odom}(\mathtt{in}\ t) &= \emptyset \\
\mathsf{idom}(\mathtt{out}\ t) &= \emptyset & \mathsf{odom}(\mathtt{out}\ t) &= \{t\}
\end{aligned}$$

Definition 1.3 Type simulation [8]. A type simulation is a relation R that satisfies the following property.

$$\begin{aligned}
(S_1, S_2) \in R \Rightarrow\ & \mathsf{idom}(S_1) \subseteq \mathsf{idom}(S_2) \\
\wedge\ & \mathsf{odom}(S_1) \supseteq \mathsf{odom}(S_2) \\
\wedge\ & \forall t \in \mathsf{idom}(S_1) \exists S_1', S_2' : (S_1 \xrightarrow{\mathtt{in}\ t} S_1' \wedge S_2 \xrightarrow{\mathtt{in}\ t} S_2' \wedge (S_1', S_2') \in R) \\
\wedge\ & \forall t \in \mathsf{odom}(S_2) \exists S_1', S_2' : (S_1 \xrightarrow{\mathtt{out}\ t} S_1' \wedge S_2 \xrightarrow{\mathtt{out}\ t} S_2' \wedge (S_1', S_2') \in R)
\end{aligned}$$

Definition 1.4 Subtyping[3]. $S_1 \leq S_2$ iff there exists a type simulation R such that $(S_1, S_2) \in R$.

Definition 1.5 Duality.

$$\begin{aligned}
\overline{\mathtt{in}\ t} &= \mathtt{out}\ t & \overline{\mathtt{out}\ t} &= \mathtt{in}\ t \\
\overline{S_1.S_2} &= \overline{S_1}.\overline{S_2} & \overline{S_1|S_2} &= \overline{S_1}|\overline{S_2} \\
\overline{\mu v.S} &= \mu v.\overline{S} & \overline{v} &= v,\ v \text{ a type variable} \\
\overline{\mathtt{end}} &= \mathtt{end}
\end{aligned}$$

[3] This is an extension of the host language's subtyping relation to provide subtyping over session types.

$$\frac{}{S \equiv S}\ (\text{Refl}) \qquad \frac{\mathsf{idom}(S_1) = \mathsf{idom}(S_2)}{((S.S_1)|(S.S_2)) \equiv (S.(S_1|S_2))}\ (|\text{Dist} \leftarrow)$$

$$\frac{S_1 \equiv S_2}{S_2 \equiv S_1}\ (\text{Sym}) \qquad \frac{}{((S_1.S)|(S_2.S)) \equiv ((S_1|S_2).S)}\ (|\text{Dist} \rightarrow)$$

$$\frac{S_1 \equiv S_2 \quad S_2 \equiv S_3}{S_1 \equiv S_3}\ (\text{Trans}) \qquad \frac{S_1 \equiv S_1'}{(S_1|S_2) \equiv (S_1'|S_2)}\ (|\text{Cong})$$

$$\frac{w \notin \mathsf{FN}(S)}{\mu v.S \equiv \mu w.(S[w/v])}\ (\mu\text{Ren}) \qquad \frac{S_1 \equiv S_1'}{(S_1.S_2) \equiv (S_1'.S_2)}\ (.\text{Cong} \leftarrow)$$

$$\frac{}{\mu v.S \equiv S[\mu v.S/v]}\ (\mu\text{Exp}) \qquad \frac{S_2 \equiv S_2'}{(S_1.S_2) \equiv (S_1.S_2')}\ (.\text{Cong} \rightarrow)$$

$$\frac{}{(S|S) \equiv S}\ (|\text{Idem}) \qquad \frac{S \equiv S'}{\mu v.S \equiv \mu v.S'}\ (\mu\text{Cong})$$

$$\frac{}{(S_1|S_2) \equiv (S_2|S_1)}\ (|\text{Comm})$$

$$\frac{}{((S_1|S_2)|S_3) \equiv (S_1|(S_2|S_3))}\ (|\text{Assoc}) \qquad \frac{v \notin \mathsf{FN}(S_1)}{\mu v.(S_1.S_2) \equiv (S_1.\mu v.(S_2[(S_1.v)/v]))}\ (.\text{Rot} \rightarrow)$$

$$\frac{}{((S_1.S_2).S_3) \equiv (S_1.(S_2.S_3))}\ (.\text{Assoc}) \qquad \frac{v \notin \mathsf{FN}(S_2)}{\mu v.(S_1.S_2) \equiv (\mu v.(S_1[(v.S_2)/v]).S_2)}\ (.\text{Rot} \leftarrow)$$

Fig. 5. Rules for equivalence

$$\frac{S \equiv (a.S')}{S \xrightarrow{a} S'}\ (\text{Cont}) \qquad \frac{S_1 \xrightarrow{a} S_1'}{(S_1|S_2) \xrightarrow{a} S_1'}\ (|\text{Elim} \leftarrow) \qquad \frac{S_2 \xrightarrow{a} S_2'}{(S_1|S_2) \xrightarrow{a} S_2'}\ (|\text{Elim} \rightarrow)$$

Fig. 6. Rules for continuation

Definition 1.6 Compatibility.

$$T \bowtie S \iff \overline{T} \leq S$$

i.e. T is defined as compatible with S iff its complement is a subtype of S.

In order to preserve compatibility between two peers in states where both inputs and outputs are permitted, we impose the *safe directionality* property on all valid

sessions. The safe directionality property is justified in Appendix A.

Definition 1.7 Safe directionality. A session S is safe-directional iff

$$\begin{aligned}&(\mathsf{idom}(S) \neq \emptyset \wedge \mathsf{odom}(S) \neq \emptyset) \rightarrow \\ &\forall t_o \in \mathsf{odom}(S) \exists S' : S \xrightarrow{\texttt{out}\ t_o} S' \wedge S \leq S' \\ \wedge\ &\forall t_i \in \mathsf{idom}(S) \exists S' : S \xrightarrow{\texttt{in}\ t_i} S' \wedge S' \leq S\end{aligned}$$

1.4 Type Mutation and Linearity

Throughout this paper, we assume a statically typed language. However, session type theory [10] states that after a session has performed a communication action, its type must automatically mutate to the session's continuation type relative to the action that has taken place. Most statically typed languages do not permit a variable's type to mutate under any circumstances, although some do allow for a variable to be overridden by one with the same name but a more restrictive scope. This seems to be the only practical way to simulate type 'mutation', but the requirement to create a new scope after every communication operation would severely restrict the structure of a program. So we adopt the strategy of introducing a new session variable after each communication action.

After we have used a session variable (i.e. by sending or receiving over it), it becomes invalid. This means that any further use of the variable is an error and would violate our typing system. A variable with such a constraint imposed upon it is known as [22] a *linear variable*, and any program that satisfies this property is said to satisfy the *linearity constraint*. We have developed a prototype tool to check linear usage of session values [3].

1.5 Closing a Session

In order to ensure the correct behaviour of the program, we impose the following constraints on the operation of closing a session. Sessions of type **end** must close their session by performing the `close` operation on the session. Furthermore, sessions of any other type may not close. The second constraint is trivial to enforce, but we may enforce the first constraint by asserting that for each statement a that assign to a session `s` of type **end**, there must exist a statement c of the form `s.close()` such that

$$c \text{ pdom } a$$

i.e. c postdominates [1] a. Intuitively this means that all sessions that are scheduled to close (by a communication operation resulting in a session of type **end**) are guaranteed to close by the control flow of the program, provided the program is not interrupted, e.g. by the operating system.

The *definite termination* property states that only sessions of type **end** may be closed. This property ensures synchrony between the communicating processes.

```
component filter {
  provide output<stream char>;
  require input<stream char>;
}
```

Fig. 7. An example of a Darwin component type (courtesy [14])

$$
\begin{array}{lll}
SS & ::= & S\texttt{ := }S\texttt{.send(}EX\texttt{)} \quad \text{(Send)} \\
 & | & S\texttt{ := }S\texttt{.receive(}EX\texttt{)} \quad \text{(Recv)} \\
 & | & S\texttt{ .close()} \quad \text{(Close)} \\
 & | & \lambda\texttt{ (}S\texttt{ (,}S\texttt{)}*\texttt{) := }S \quad \text{(Lambda)}
\end{array}
$$

Fig. 8. Syntax of session statements. Greyed out syntax is not present in the input data.

2 Related Work

This work's main underpinning, session typing, was first introduced by Honda [10]. This work also introduced session type inference for the π-calculus. Dezani-Ciancaglini et al [6] brought session types to the imperative world with the language Moose. They [5] later expanded upon this work with a notion of compatibility [8].

Other means of specifying and verifying protocols for compatibility include finite state automata (including interface automata [4] and choreography [7]), channel contracts [11] and component interfaces [2].

Ninja provides a component model similar to that of the Unified Modeling Language [16] or architecture description languages such as Darwin [14]. While the UML component model largely deals in the abstract, permitting any form of communication such as a streaming model, shared memory model or procedure calls, Ninja's model, similar to Darwin's, restricts communication to a streaming model using the provided communication channels. Darwin's communication channels have a simple notion of typing as shown in the example component type of Figure 7, however the session typed nature of Ninja's channels affords a greater deal of flexibility.

3 Derivation and Canonicity

This section provides a high level description of our inference algorithm's derivation steps. As our algorithm is language independent, the control structure is defined by the language. In particular, the host language should define the following:

Stmts	set of session program statements
$\vdash$	type assignment for expressions
EX	syntax for expressions
S	syntax for session variables

The syntax for program statements that operate on sessions is, however, defined by the syntax given in Figure 8. Our algorithm supports an unbounded number of

concurrent sessions.

Definition 3.1 Choice composition. The choice composition operator $\Big|$ is defined nondeterministically as follows.

$$\Big|T = \begin{cases} (t\Big|(T \setminus \{t\})), & |T| \geq 2 \wedge t \in T \\ t, & T = \{t\} \end{cases}$$

Definition 3.2 Canonicity. In the following, a is an action.

(i) $(S_1|S_2)$ is canonical if sessions S_1 and S_2 are canonical, $\mathsf{idom}(S_1) \cap \mathsf{idom}(S_2) = \mathsf{odom}(S_1) \cap \mathsf{odom}(S_2) = \emptyset$, $S_1 \not\equiv \texttt{end}$ and $S_2 \not\equiv \texttt{end}$.

(ii) $(a.S)$ is canonical if S is canonical.

(iii) $(S_1.S_2)$ is *not* canonical if S_1 is not an action.

(iv) $\mu v.S$ is canonical if S is canonical and $v \in \mathsf{FV}(S)$.

(v) `end` is canonical.

(vi) v is canonical.

(vii) a is *not* canonical.

We begin by rewriting all statements of form $[\![s\texttt{.send}(e)]\!]$ to $[\![s\ \texttt{:=}\ s\texttt{.send}(e)]\!]$; and all statements of form $[\![s\texttt{.receive}(e)]\!]$ to $[\![s\ \texttt{:=}\ s\texttt{.receive}(e)]\!]$. We proceed to convert session statements to single static use [13] form. The rules given in Figure 9 are then applied to assign a type to each session variable by solving for $\Delta = \emptyset$ where Γ contains language-specific typing information for the current context. Each well-formed type must have a *canonical* form as described in Definition 3.2, which is equivalent to the original derived type according to the equivalence rules given in Figure 5. If any type is not well-formed, i.e. it does not have an equivalent canonical form, the inference algorithm fails. After the canonical form for each session type is derived, we eliminate λ statements by first globally replacing any session variable appearing on the left hand side of a λ statement with the session variable named on the right hand side, then removing the λ statements themselves. Note that session variables retain the type assigned to them before λ statements were eliminated.

3.1 Justification

This section gives reasoning behind parts of our derivation process given above.

Canonicity rule (i) ensures that no two alternatives in a choice construct may present the same choices. This rule ensures the deferment of such choices to the last possible moment. This reflects the restrictions imposed on the communicating process, namely that a process may only choose which branch it takes on the basis of the type of the variable it sends or receives, and not any other information. In the process of applying equivalence rules to a session type in order for it to conform with canonicity rule (i), equivalence rule ($|\text{Dist} \leftarrow$) will be most frequently employed. This rule prevents the situation shown in Section 1 where two distinct branches of a session type are initially distinguished by the types of their inputs. There is no need to impose such a rule on branches which are initially distinguished

$$\frac{[\![s.\texttt{close}()]\!] \in \mathsf{Stmts}}{\Gamma, \Delta \vdash s : \texttt{end}} \text{ (Close)}$$

$$\frac{[\![s'{:=}s.\texttt{send}(v)]\!] \in \mathsf{Stmts} \quad \Gamma \vdash v : t_v \quad \varphi \text{ fresh} \qquad \Gamma, \Delta[s \mapsto \varphi] \vdash s' : t_{s'} \quad s \notin \text{dom } \Delta}{\Gamma, \Delta \vdash s : \mu\varphi.(\texttt{out } t_v.t_{s'})} \text{ (Send)}$$

$$\frac{[\![s'{:=}s.\texttt{receive}(v)]\!] \in \mathsf{Stmts} \quad \Gamma \vdash v : t_v \quad \varphi \text{ fresh} \qquad \Gamma, \Delta[s \mapsto \varphi] \vdash s' : t_{s'} \quad s \notin \text{dom } \Delta}{\Gamma, \Delta \vdash s : \mu\varphi.(\texttt{in } t_v.t_{s'})} \text{ (Recv)}$$

$$\frac{[\![\lambda(s_1, s_2, \ldots, s_n){:=}s]\!] \in \mathsf{Stmts} \quad \varphi \text{ fresh} \qquad \forall 1 \leq i \leq n : \Gamma, \Delta[s \mapsto \varphi] \vdash s_i : t_i}{\Gamma, \Delta \vdash s : \mu\varphi.(\Big|\{t_i : 1 \leq i \leq n\})} \text{ (Lambda)}$$

$$\frac{\Delta(s) = \varphi}{\Gamma, \Delta \vdash s : \varphi} \text{ (Abbrv)}$$

Fig. 9. Type inference rules

by the types of their outputs, as a communicating process may simply accept both value types at this point.

4 Algorithm

This section supplies a concrete description of our type inference algorithm suitable for implementation. Our algorithm is implemented in three stages. For the purpose of illustration we shall use a simplified version of Ninja-C++ called $\mathcal{L}_\mathsf{N}$ whose syntax contains only `if`, `while` and session communication statements with the symbol $*$ substituted for boolean expressions and expression types substituted for all other expressions and whose control flow is defined in the obvious way. It is possible to translate a Ninja-C++ program written in C++ into $\mathcal{L}_\mathsf{N}$ by converting `for` loops into `while` loops in the usual way, removing all variable declarations, removing all statements without a counterpart in $\mathcal{L}_\mathsf{N}$ and replacing all primitive values with their types. Note that $\mathcal{L}_\mathsf{N}$ does not include invocations because we are not inferring the type of the channel; it may have any type less specific than the participant's dual and more specific than the invoker's, and compatibility between participants and invokers is achieved by upcasting the return value from the `invoke` method into the appropriate type. Our language supports an unbounded number of concurrent sessions.

```
if (*) {
  s1 := s1.send(int) ;
  s1 := s1.receive(int)
} else {
  s1 := s1.send(long) ;
  s1 := s1.receive(long)
} ;
s1 := s1.send(bool)
```

Fig. 10. Simple $\mathcal{L}_{\mathsf{N}}$ program.

```
λ(s2, s3) := s1 ;
if (*) {
  s4 := s2.send(int) ;
  s6 := s4.receive(int)
} else {
  s5 := s3.send(long) ;
  s6 := s5.receive(long)
} ;
s7 := s6.send(bool)
```

Fig. 11. Simple $\mathcal{L}_{\mathsf{N}}$ program after SSU applied.

4.1 Stage 1: Static Single Use

The first step is to ensure that no session variable is reused more than is necessary. This is different from the linearity constraint mentioned in Section 1.4; what we would like to do here is to detect legitimate, linear programs that reuse session variables instead of using a fresh variable wherever possible, meaning that our inference algorithm would generate too general a session type. In the most extreme case, only one session variable is used throughout an entire procedure (note that this is the starting point of our derivation algorithm). Thus the program's communication statements are first converted to SSU [13] form. Figure 10 shows a program in $\mathcal{L}_{\mathsf{N}}$ with liberal reuse of session types, and Figure 11 shows the same program after SSU has been applied to it.

4.2 Stage 2: Graph Building

After obtaining the SSU form of the program, we then build a graph of the session transitions contained within the program using its communication statements. The function g that builds this graph is shown in Figure 12, assisted by the *unification mapper* f_G shown in Figure 13. The goal of this function is twofold:

- to extract all communication actions and collect them into a graph with arcs between source and target session variables;
- for variable assignments, ensure that the source and target sessions receive the same type (this is the purpose of the δ function built by f_G). The helper function ϵ (Figure 13) assists in this by providing a means for a given set of variables to

$$
\begin{aligned}
g(p) &= f_G(G, \delta) \\
\text{where } (G, \delta) &= f_P(p) \\
f_P(s' \texttt{ := } s \texttt{ .send(}t\texttt{)}) &= \\
((\{\{s\}, \{s'\}\}, \{(\{s\}, \{s'\}, \texttt{out } t)\}, \emptyset), \lambda x.x) & \\
f_P(s' \texttt{ := } s \texttt{ .receive(}t\texttt{)}) &= \\
((\{\{s\}, \{s'\}\}, \{(\{s\}, \{s'\}, \texttt{in } t)\}, \emptyset), \lambda x.x) & \\
f_P(s' \texttt{ := } s) &= ((\emptyset, \emptyset, \emptyset), \epsilon(\{s, s'\})) \\
f_P(s \texttt{ .close()}) &= ((\emptyset, \emptyset, \{\{s\}\}), \lambda x.x\})) \\
f_P(\texttt{if (*) \{ } p_1 \texttt{ \} else \{ } p_2 \texttt{ \}}) &= f_P(p_1 \texttt{ ; } p_2) \\
f_P(\texttt{while (*) \{ } p \texttt{ \}}) &= f_P(p) \\
f_P(s \texttt{ .recv\_choice \{ } c \texttt{ \}}) &= f_C(c) \\
f_P(\lambda \texttt{ (} s_1 \texttt{ , ... , } s_n \texttt{) := s}) &= ((\emptyset, \emptyset, \emptyset), \epsilon(\{s, s_1, ..., s_n\})) \\
f_P(p_1 \texttt{ ; } p_2) &= \\
((n_1 \cup n_2, e_1 \cup e_2, a_1 \cup a_2), \delta_1 \circ \delta_2 \circ \delta_1) & \\
\text{where } ((n_1, e_1, a_1), \delta_1) &= f_P(p_1) \\
((n_2, e_2, a_2), \delta_2) &= f_P(p_2) \\
f_C(\texttt{case } t \texttt{ : } p) &= f_P(p) \\
f_C(c_1 \texttt{ ; } c_2) &= \\
((n_1 \cup n_2, e_1 \cup e_2, a_1 \cup a_2), \delta_1 \circ \delta_2 \circ \delta_1) & \\
\text{where } ((n_1, e_1, a_1), \delta_1) &= f_C(c_1) \\
((n_2, e_2, a_2), \delta_2) &= f_C(c_2)
\end{aligned}
$$

Fig. 12. Graph building function g.

$$
\begin{aligned}
f_G((N, E, A), \delta) = \; & (\{\delta(n) | n \in N\}, \\
& \{(\delta(n), \delta(n'), e) | (n, n', e) \in E\}, \\
& \{\delta(a) | a \in A\}) \\
\epsilon(S)(x) = \; & \begin{cases} x \cup S, & x \cap S \neq \emptyset \\ x, & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 13. Unification mapper and helper function ϵ.

receive the same type.

After the graph is built, the definite termination property is checked. The definite termination property can be expressed as follows for a graph $G = (N, E, A)$:

$$\forall a \in A : \nexists n_2 \in N, e : (a, n_2, e) \in E$$

Note that if multiple sessions are used concurrently, the graph will be composed of disjoint subgraphs. These graphs are independent and will not affect one another except possibly during *safe* merging operations in stage 3.

4.3 Stage 3: Graph Simplification and Translation

At this stage we must first process the graph in order to identify and merge nodes such that semantics are preserved. Furthermore we wish to identify invalid graphs.

To begin with, let us define a notion of node equivalence within our graph.

Definition 4.1 Node equivalence within a graph.

$$\begin{aligned}\mathrm{eq}(ns, c, (N, E, A)) \longleftrightarrow\ & ns \in c \vee |ns| \leq 1 \vee \\ & (\bigwedge\{\mathrm{eq}(\{n' | n \in ns \wedge (n, n', e) \in E\}, \\ & c \cup \{ns\}, (N, E, A)) \\ & | n \in ns \wedge (n, , e) \in E\} \\ & \wedge (ns \subseteq A \vee A \cap ns = \emptyset))\end{aligned}$$

$$n_1 \equiv_G n_2 \leftrightarrow \mathrm{eq}(\{n_1, n_2\}, \emptyset, G)$$

Definition 4.2 Applying a substitution function. To apply a substitution function δ, we replace the current graph G with the result of unification mapper $f_G(G, \delta)$, where f_G is defined in Figure 13.

We may unify nodes provided that they are node equivalent, according to Definition 4.1. This allows us to simplify graphs with multiple convergent arcs with the same label leading to a single node. For each pair of nodes n_1 and n_2 in our graph G such that $n_1 \equiv_G n_2$, we apply the substitution function $\epsilon(n_1 \cup n_2)$.

A second case we must deal with is divergence. Should a graph have many divergent arcs with the same label leading to nodes $n_1, n_2 ... n_n$, we must replace these nodes and any dependent subgraph with a single node n and associated subgraph such that $\forall i \in \{1...n\}, \tau(n) \leq \tau(n_i)$, where τ is the session type building function defined in Figure 14. The initial processing may be achieved by treating our session graph as a NFA, converting it into a DFA using the subset construction and rejecting any graph that does not satisfy this property. This transformation is sound as it has been proven [18] that each NFA has an equivalent DFA (accepting the same language or, in our case, sequence of communication actions) which translates directly to trace soundness.

Before we check this property we must convert node labels in the DFA from sets of sets to sets in order to make it consistent with the NFA. This is done by applying

$$\tau(n, G) = \tau_S(\{n\}, G, \emptyset)$$

$$\tau_S(ns, (N, E, A), \delta) = \begin{cases} \delta(n), & n \in ns \cap \operatorname{dom} \delta \\ \mu\varphi.\Big| \begin{array}{l} \{(a.\tau_S(\{n' : n \in ns \wedge (n, n', a) \in E\}, \\ \quad (N, E, A), \delta[n \mapsto \varphi : n \in ns])) \\ \quad : n \in ns \wedge (n, \_, a) \in E\} \end{array}, & \begin{array}{l} ns \cap A = \emptyset \\ \varphi \text{ fresh} \end{array} \\ \texttt{end}, & \text{otherwise} \end{cases}$$

Fig. 14. Session type building function τ.

the substitution function

$$\delta(x) = \bigcup x$$

A simple way of verifying the above property is to do so 'superficially' between each node in the DFA graph and each corresponding component node in the original graph, as formulated below.

Definition 4.3 Superficial subtyping. A type graph $H = (N_H, E_H, A_H)$ is a superficial subtype of a type graph $G = (N_G, E_G, A_G)$ iff:

$$\begin{aligned} \forall n_h \in N_H, n_g \in N_G : n_g \subseteq n_h \implies & \mathsf{idom}(s_g) \subseteq \mathsf{idom}(s_h) \\ & \wedge\ \mathsf{odom}(s_g) \supseteq \mathsf{odom}(s_h) \\ \text{where } s_g = \tau(n_g, G) & \\ s_h = \tau(n_h, H) & \end{aligned}$$

Note that in practice, the superficial subtyping property implies that each node must have an identical set of input types, and there are no restrictions on output types.

If the DFA nodes have overlapping subsets, which is entirely possible based on the structure of our program, we will not be able to type those session variables that appear in two or more nodes, as each session variable must have a single type. Thus for each node in the original graph we must merge all nodes in the resultant graph containing that node; i.e. for each original node n we apply the substitution function

$$\delta(n_h) = \begin{cases} \bigcup\{m | m \in N_H \wedge n \in m\}, & \text{if } n \in n_h \\ n_h, & \text{otherwise} \end{cases}$$

If the new graph no longer satisfies the superficial subtyping, definite termination or safe directionality property given above, we must reject it.

Note that the merging of overlapping subsets preserves trace soundness but not trace completeness. This is a small concession, and because we applied SSU to the program before simplifying the graph, it is also the smallest possible concession that we can make.

We may now extract the session types from our graph by employing the τ function shown in Figure 14 and using equivalence rules, in particular μExp and congruence, in order to eliminate unnecessary μ operators.

5 Example

This section presents an example of how our algorithm is used to derive session types. We start with the following communication procedure

```
void server(session s) {
  while (1) {
    s = s.receive(Req1(x));
    if (x%2) {
      s = s.send(x+1);
      s = s.receive(x);
      s = s.send((char) x%256);
    } else {
      s = s.send(x-1);
      s = s.receive(x);
      s = s.send((long) x<<16);
    }
  }
}
```

Firstly we convert this program to $\mathcal{L}_{\mathsf{N}}$ by removing statements and simplifying:

```
while (*) {
  s1 := s1.receive(Req1) ;
  if (*) {
    s1 := s1.send(int) ;
    s1 := s1.receive(int) ;
    s1 := s1.send(char)
  } else {
    s1 := s1.send(int) ;
    s1 := s1.receive(int) ;
    s1 := s1.send(long)
  }
}
```

We proceed to stage 1, converting to SSU form:

```
while (*) {
  s2 := s1.receive(Req2) ;
  λ(s3, s4) := s2 ;
  if (*) {
    s5 := s3.send(int) ;
    s6 := s5.receive(int) ;
    s1 := s6.send(char)
  } else {
```

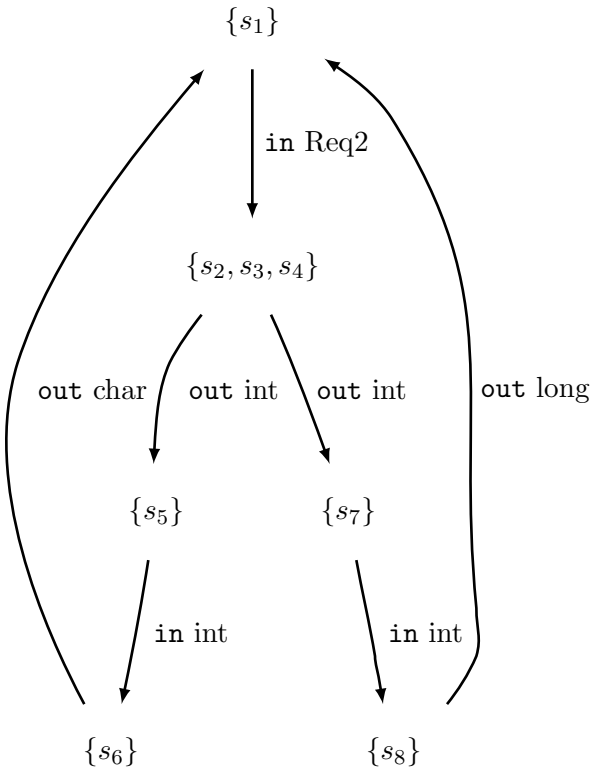

Fig. 15. Result of graph building function f_G applied to Example 1.

```
    s7 := s4.send(int) ;
    s8 := s7.receive(int) ;
    s1 := s8.send(long)
  }
}
```

Applying the graph building function f_G we obtain the graph shown in Figure 15. This graph has no accepting states so the definite termination property vacuously holds.

The graph has no recursively equal nodes for us to unify, so we proceed to DFA building using the subset construction, giving us the graph shown in Figure 16. In this graph, each node is a disjoint subset of the set of sessions, so our substitution function has no effect. Applying the τ function to our graph to produce a Ninja session type, we deduce the following overall type assignment for s_1:

$$s_1 : \mu t.\texttt{in}\ \text{Req2}.\texttt{out}\ \text{int}.\texttt{in}\ \text{int}.(\texttt{out}\ \text{char}.t|\texttt{out}\ \text{long}.t)$$

6 Sessions in C++

This section shall describe how the Ninja language has been adapted to standard C++ in our language Ninja-C++, without the use of any special compilers or language extensions.

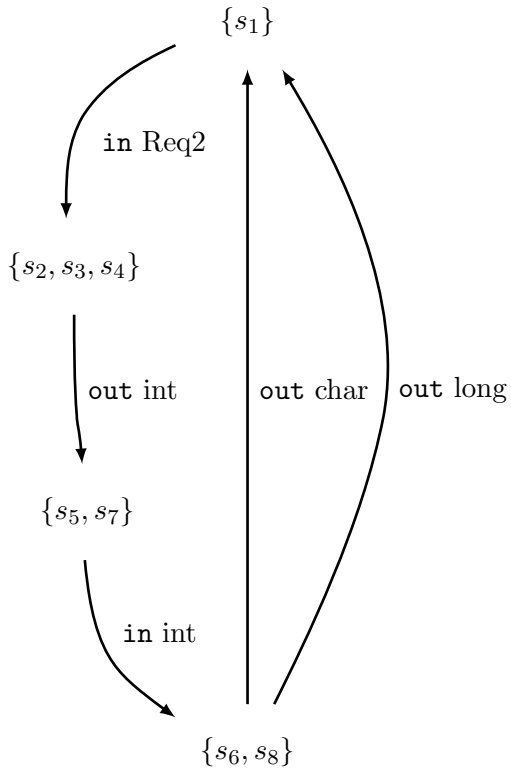

Fig. 16. Result of subset construction applied to Figure 15.

6.1 Sessions and Channels

In Ninja-C++ sessions are represented as a hierarchy of template instantiations, as the C++ template mechanism allows us to specify a user-defined type hierarchy. As we shall see, the template-based representation can express almost every session type in our algebraic representation, modulo equivalence, discounting some restrictions on choice.

We distinguish between Ninja *actions* and *sessions*. An action is a primitive communication step, such as **in int** (of the form $D\ VT$ from Figure 4), whereas a session is a fully specified session which may include sequential composition, choice etc. Actions in Ninja-C++ shall take the form **in**<T> or **out**<T>, where T is the primitive data type to be sent or received across the channel.

For sessions, the two main constructs that we must represent are sequential composition (`.` in Ninja) and choice (`|` in Ninja). Sequential composition will compose an action with a session (its continuation), so our best choice of representation is **seq**<A,S>, A being the action and S the session. For the choice construct we compose sessions via a tuple-style representation of the form **choice**<S1,S2,...,Sn>. The constraints implied by these template declarations allow for relatively trivial derivation of the input domain, output domain and continuation type of a particular type at each stage and, for this reason, provide the basis for additional constraints imposed on derived types as we shall see.

Frequently when designing session types, we must be able to create recursive types. This will commonly occur when we would like to represent a loop in our

```
struct s;
typedef seq<in<int>,call<s> > r;
struct s { typedef r t; };
```

Fig. 17. A recursive session type representing an infinite stream of `int` inputs

session typed code (for example a request-response loop, or a computation which may produce an arbitrary number of responses). The most obvious way of creating a recursive type in C++ (that is, defining a type in terms of itself in a **typedef** statement) will not work, because the language prevents such a definition. However an incompletely-defined type may be referred to in a template instantiation. This allows us to specify a three-stage protocol that may be used to define a recursive type. Firstly, an incomplete struct s is defined. Secondly, the recursive session type r is defined using a **typedef**. Wherever a recursive reference is required, the special instantiation **call**<s> is used. Thirdly, s is fully defined, with an internal typedef t that is defined to be r. An example of such a definition is shown in Figure 17.

What we have done in the previous paragraph is establish an *isorecursive type system* [17]. As opposed to the equirecursive type system of Ninja, where a recursive type and references to the recursive type are equivalent via the (μExp) rule given in Figure 5, in our isorecursive type system we have established an isomorphism between the 'rolled' reference type **call**<s> and the 'unrolled' type r. The 'unroll' operation is carried out automatically during the computation of the continuation type of a particular session type if it is found to be of the form **call**<s>. In this case there is no inverse mapping from unrolled types to rolled types; we do not require one here, but it would be trivial to define one in order to make this a 'true' isorecursive type system.

The primitives `invoke`, `send`, `receive`, `newchannel` and `spawn` are implemented, as in Ninja, as methods of the applicable classes, i.e. sessions (`send`, `receive`), channels (`invoke`), participants (`spawn`). The `newchannel` primitive is presented as a type constructor for the channel type.

We must define types for sessions and channels themselves. We have defined a type **session**<S> for sessions, where S is the session type. Similarly we have **channel**<S> for channels.

6.2 Participants

Each participant comprises:

- a list of its channels, including information regarding whether the channel is linear, shared or invokable;
- for those channels which are linear or shared, an implementation of a communication procedure for that channel;
- for those channels which are invokable, a variable which will store the channel.

and provides the following functionality:

- a constructor which is provided with a sequential list of channels in the order provided by its definition;

```
struct part_base {

  channel<s1> *ch1;

  void ch2(session<s2> s) {
    ...
  }

};

typedef participant<part_base,
  dual_channel<s1>, &part_base::ch1,
  linear_channel<s2>, &part_base::ch2
> part;
```

Fig. 18. An example of a skeleton participant

- a `spawn` method which spawns the participant.

Participants are implemented as a `participant` template which is parameterised over the types of its channels and the names of the relevant communication procedures and channel fields. This allows us to perform compile-time type checking of channels supplied to the participant.

The `spawn` primitive in Ninja takes an argument indicating the 'location' of the participant. Normally this means the CPU core on which it shall run. Obviously the specification of a location is implementation-specific, but in order to allow for portable programs to be written, all implementations must provide a default location. For a particular implementation, this may mean a particular core, or it may mean that the underlying operating system should select one automatically. In any case, the default location is given in the constant os::default_location.

A participant's communication procedures and channel variables are encapsulated by making them non-static members of their own class, known as the participant implementation class. The name of this class is supplied as a parameter to `participant`, which will declare it as a base class. Note that we cannot have the participant implementation class be a subclass of the `participant` instantiation. This is because it would entail that the implementation class be defined in terms of the participant class (as it is a base class). Recall that the participant class is parameterised over the implementation class's fields and methods. So we have a circular reference, which is not possible in the C++ language. `participant`'s template parameters will thus comprise its base class (the implementation class) and the list of channels.

An example of a skeleton participant is shown in Figure 18.

6.3 A Note on Session Variable Types

As previously mentioned, each session variable must be fully specified with its session type. It is unfortunate that the C++ language does not provide us with the

facility of automatically deducing the session variable's type, even though it has all the information available to do so. The most recent draft of the C++ standard [12] provides for an `auto` specifier for variable declarations (section 7.1.5.4) which deduces the type of a variable from the type of its initialiser. This would be ideal for our purposes here, but since the document is still in draft, no compiler implements this feature yet, and we have to make do with what we have.

7 Implementation

Our prototype implementation of this algorithm covers stages 2 and 3 of the algorithm described in Section 4, with two crucial differences:

- As Ninja-C++ does not currently take into account session subtyping as described in Definition 1.4, an invoker's communications must produce the exact same session type via our algorithm as the dual of the corresponding communication procedure for them to be compatible.
- It only performs a simplified version of the subset construction, and does not check the superficial subtyping property for minimised graphs.

It is a C++ program transformation using the ROSE [19] source-to-source translator framework. The transformation takes an untyped Ninja-C++ program as input, and generates a compilable typed program as output.

The first step in implementing the algorithm is to create an untyped version of Ninja-C++. Creating an untyped version of the language entails creating versions of the session and channel templates that do not take session type parameters. The two use cases for our type inference system are deriving intermediate session types, and deriving full session and participant information. Thus we must have two variants of our untyped implementation; for the first, only session and channel are untyped (known as the untyped sessions variant); for the second, everything is untyped (known as the untyped participants variant).

The implementation of the algorithm is used to automatically assign types to sessions, channels and participants. It proceeds in three stages. Firstly it uses an AST traversal to collect information about the session usages, channel invocations and participant definitions that the program uses. Information about session usages is stored in a graph-like structure, a mapping between a node and a set of arcs. Each arc stores direction and type information as well as the node the arc points to. Each node stores a set of ROSE AST variable declarations which represent the session variables that correspond to the type at that node. Information about channel invocations is stored as a mapping from channel variables (AST variable declaration for the channel) to session nodes. Information about participant definitions is stored as a mapping from the template parameter representing the channel type to the session node.

After the information has been collected, all sessions pertaining to a channel invocation (found by using the channel invocation information that has been collected, as well as by following the session usage graph) are 'flipped' and marked as dual.

Secondly, the process of unification takes place. This proceeds in two stages, which repeat execution alternately until both stages cannot modify the graph. In the first stage, we unify identical divergent paths using the subset construction. In the second stage we unify based on recursive equality.

8 Conclusions and Future Work

We have shown how our type inference system allows for a program's behaviour to be expressed as a type. We have further shown how programs can be judged to be compatible by a language's type system using their assigned types. This allows the developer greater freedom in designing client/server programs, as the compatibility between the two peers can be checked at compile time without the developer needing to compute the program's session type manually. We have also described a well-formedness constraint for session types with implicit choice that forbids session types for which a dual cannot be constructed.

Ninja-C++ does not currently decide compatibility according to Definition 1.6; instead, two session types are deemed to be compatible only if they are the exact dual of each other. Clearly, this does not afford us much flexibility. The reason for this is that any such compatibility check, being a compile-time mechanism, must take place within the language's facilities for compile-time computation. For C++, this means the template system. However, the C++ template system, despite being Turing complete [21], has insufficient expressibility for a maintainable implementation of the compatibility relation to be feasible. In order to add a dynamic layer of expressibility to the language, a compile-time extension framework can be implemented providing computed template instantiations in a functional, or semi-functional, language such as ML or Haskell. In this instance, the extension framework can be used to build a template representing a binary relation of session subtyping as described in Definition 1.4. We can then use custom type conversion operators and the Substitution Failure Is Not An Error (SFINAE) principle to facilitate substitutability and thus, by the construction of Ninja-C++, compatibility.

Ninja-C++ supports callable procedures that perform operations over session types. In order to preserve type safety, such procedures are parameterised over the remainder of the session type using C++ templates. However, our inference system does not currently infer the session type of such procedures correctly. In order to support interprocedural session type inference, the algorithm must be extended to recognise where parameterisation is necessary (i.e. the passing of session variables between procedures) and insert the correct template syntax where required.

9 Acknowledgements

This work was supported by an EPSRC PhD studentship. The authors would like to thank Nobuko Yoshida and Sophia Drossopoulou for their valuable input.

References

[1] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. Frank. Exploiting postdominance for speculative parallelization. In *High Performance Computer Architecture*, February 2007.

[2] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal component model, 2004.

[3] P. C. Collingbourne. Verification tools for multi-core programming. Master's thesis, Imperial College, London, United Kingdom, 2007.

[4] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[5] M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded Session Types for Object-Oriented Languages. In *FMCO'06*, LNCS. Springer-Verlag, 2007.

[6] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In D. Thomas, editor, *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.

[7] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 738, Washington, DC, USA, 2004. IEEE Computer Society.

[8] S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2):191–225, 2005.

[9] S. Gay, V. T. Vasconcelos, and A. Ravara. Session types for inter-process communication. TR 2003–133, Department of Computing, University of Glasgow, Mar. 2003.

[10] K. Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.

[11] G. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. D. Zill. An overview of the Singularity project. Technical report, Microsoft Research, October 2005.

[12] International Standards Organisation. Programming languages – C++, 2006-11-06.

[13] R. Lo, F. Chow, R. Kennedy, S.-M. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 26–37, 1998.

[14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical Report -86, 1989.

[16] Object Management Group. UML superstructure specification v2.1.1. Technical report, 2007.

[17] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[18] M. O. Rabbin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[19] M. Schordan and D. J. Quinlan. A source-to-source architecture for user-defined optimizations. In L. Böszörményi and P. Schojer, editors, *JMLC*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer, 2003.

[20] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. *Fundamenta Informaticæ*, 73(4), 2006.

[21] T. L. Veldhuizen. C++ templates are Turing complete.

[22] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

A Safe Directionality

The safe directionality property, as described in Section 1, is imposed on asynchronous communication models, such as Ninja, in order to preserve compatibility between peers. If both peers are in a state where both inputs and outputs are permitted, and they simultaneously send data to each other, they will both have followed different 'paths' through the session type thus risking that their respective 'believed' types for the session be incompatible. In the case of bidirectionality, we sacrifice 'path' correctness, but maintain compatibility of the believed current session types.

To see that the safe directionality property is correct for bidirectional types, we consider a session s_1 and its communicating peer s_2 such that $s_1 \bowtie s_2$. In order to derive the minimal conditions that must be imposed on s_1, we must consider the most specific s_2 such that $s_1 \bowtie s_2$; i.e. $s_2 = \overline{s_1}$. Suppose that peer p_1 of session type s_1 sends a message of type $t_1 \in \mathsf{odom}(s_1)$ simultaneously with p_2 of session type s_2 whose message is of type $t_2 \in \mathsf{odom}(s_2)$. Their session types are now respectively s_1' and s_2', where $s_1 \xrightarrow{\texttt{out}\ t_1} s_1'$ and $s_2 \xrightarrow{\texttt{out}\ t_2} s_2'$. We should now expect p_1 to be able to handle the message sent from p_2 in its new session s_1'. For this to be the case, $s_1 \leq s_1'$. Similarly, $s_2 \leq s_2'$, which may be rewritten $\overline{s_1} \leq \overline{s_1''} \Rightarrow s_1'' \leq s_1$ where $s_1 \xrightarrow{\texttt{in}\ t_2} s_1''$ by definition 1.5 and the standard session typing result:

$$S \leq T \iff \overline{T} \leq \overline{S}$$

The clearest instance of a bidirectional type that satisfies the safe directionality property is s_{min} such that $s_{min} \xrightarrow{a} s_{min}$ for all $a \in \{\texttt{in}\ t : t \in \mathsf{idom}(s_{min})\} \cup \{\texttt{out}\ t : t \in \mathsf{odom}(s_{min})\}$.