Novel Type Systems for Concurrent Programming Languages
GR/N39494/01
Final Report

Dr S. J. Gay
Department of Computing Science,
University of Glasgow

March 13, 2003

Exceptional Circumstances

This project began on 1st October 1997 for three years under grant GR/L75177, the principal investigator being Dr Simon Gay who was then at Royal Holloway, University of London. The grant was restructured in 2000 for two reasons: Dr Gay moved from Royal Holloway to the University of Glasgow; and the project student, Malcolm Hole, had been diagnosed with cancer and needed to change from full-time to part-time working after a period of sick leave. Therefore GR/L75177 was terminated on 29th February 2000, and the project was reannounced as GR/N39494 (the present grant), with Professor Steve Schneider as principal investigator, Dr Gay as co-investigator, and a termination date of 31st March 2002. Subsequently, Mr Hole required another lengthy period of medical treatment and the grant was extended until 31st December 2002.

This report describes the results of the entire project (under both GR/L75177 and GR/N39494), although the accompanying financial reports are for GR/N39494 alone, as a final report on GR/L75177 was submitted in May 2000. The report has been written by Dr Gay, the original principal investigator.

1 Background/Context

Types are an important aspect of high-level programming, and the increasing emphasis on static (compile-time) typechecking is a significant contribution to the development of modern, safe programming languages. Type information allows specifications of desirable program behaviour to be expressed concisely and verified automatically. For example, in a simply typed language, a function type specifies that if the parameters have certain types then the result has a certain type; straightforward syntactic analysis can verify that a given function definition satisfies this property. Convenient compromises between expressiveness and tractability have emerged, and sophisticated practical type systems have been developed, initially in the realm of functional languages but more recently for object-oriented languages.

In concurrent and distributed programming, behaviour is much richer and there is a correspondingly greater range of properties which we would like to verify. The type-theoretic approach to verification of concurrent systems is just one approach, but it is attractive because typed sequential programming is already established and we might hope to smoothly extend the type system of a sequential language by adding typing constructs corresponding to communication or distribution primitives.

The pi calculus is a particular formalism for concurrent systems which has been used for modelling and analysing a wide range of applications, but also as a vehicle for research into type systems for concurrent programming. In the pi calculus, point-to-point communication between concurrent components takes place along named channels. Two key features are responsible for much of its expressivity: channels are first-class values, and can be transmitted along channels; and new channels can be created dynamically. In terms of investigating type systems for concurrent programming, the pi calculus can be viewed as a model of a concurrent programming language which uses this style of communication.

It is straightforward to assign types to channels (for example, specifying that a particular channel is always used to carry messages of a particular datatype) and statically check that these constraints are obeyed throughout a system. From this starting-point, research on type systems for the pi calculus has developed in several directions. One approach is to specify constraints on the relationship between communications on different channels (for example, specifying partial orders among channels and checking that these orders are respected by the actual communications in a program) in order to eliminate deadlock or race conditions. Another possibility is to constrain the use of individual channels (for example, specifying that certain channels can only be used once, by means of a *linear* type system) in order to enable optimized compilation of common programming idioms. A third approach, described by the general title of *session types*, is to provide more sophisticated specifications of the communication capability of a single channel. For example, instead of specifying that a channel carries integers, we might specify that it is used first to send an integer, then to receive a boolean value, then to receive an integer. Branching and recursion allow richer sequences of messages to be constructed. Session types can be used to specify complex communication protocols, such as those typically found in client-server systems, and static typechecking can be used to verify that the implementation of a client or server observes the required protocol.

The main contribution of the project has been to further develop the theory of session types in the pi calculus. We have reformulated the original version of session types, to remove syntactic irregularity and make session channels fully first-class values. We have developed a notion of subtyping for session types, and shown that it increases the expressiveness and flexibility of specifications of client-server systems. We have investigated bounded polymorphism in session types, and obtained still further expressiveness. All of these developments in the type system have been supported by rigorous proofs, with respect to a formal semantics of the pi calculus, that the run-time use of channels corresponds to the compile-time specification.

Establishing the desirable theoretical properties of a type system requires lengthy combinatorial proofs by induction over syntactic structures. We have investigated the use of automatic theorem proving technology for reasoning about type systems in the pi calculus, particularly in the presence of linear typing features which greatly increase the complexity.

The research has been presented in conference papers and technical reports. A PhD thesis and a journal paper will be submitted during 2003.

2 Key Advances and Supporting Methodology

The essential idea of session types, developed in a series of papers by Honda *et al.*, is that the type of a communication channel describes a protocol. A session type specifies the sequence and structure of messages, and permits much richer behaviour than a simple association of a datatype with a channel.

For example, consider a server which provides a single operation, addition of integers, and specifies the following protocol.

The client sends two integers. The server sends an integer which is their sum, then closes the connection.

The corresponding session type, from the server's point of view, is

$$S = ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\mathsf{End}$$

in which ? means *receive*, ! means *send*, dot (.) is sequencing, and End indicates the end of the session. This type captures the parts of the specification which we can reasonably expect to verify statically. The server communicates with a client on a channel called c; we think of the client engaging in a *session* with the server, using the channel c for communication.

An implementation of the server in the pi calculus (here we use a more verbose syntax to increase readability) looks like this:

```
receive $x$ on $c$
receive $y$ on $c$
send $x + y$ on $c$
```

If the programmer specifies that channel c has type S, then it is possible to check statically (at compile-time) that the implementation of the server follows the protocol.

Interchanging ? and ! yields the type describing the client side of the protocol:

$$\overline{S} = !\mathsf{Int}.!\mathsf{Int}.?\mathsf{Int}.\mathsf{End}$$

and a client implementation uses the server to add two particular integers; the *code* may use x but cannot use the channel c except to close it.

```
send 2 on $c$
send 3 on $c$
receive $x$ on $c$
code
```

Again it is possible to check statically that this client follows the protocol specified by $\overline{S}$.

More interesting protocols involve branching. If we add a negation operation to the server:

> The client selects one of two commands: add or neg. In the case of add the client then sends two integers and the server replies with an integer which is their sum. In the case of neg the client then sends an integer and the server replies with an integer which is its negation. In either case, the server then closes the connection.

The corresponding session type, for the server side, uses the constructor & (*branch*) to indicate that a choice is offered.

$$S = \&\langle \mathsf{add}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\mathsf{End},\ \ \mathsf{neg}\colon ?\mathsf{Int}.!\mathsf{Int}.\mathsf{End}\rangle$$

Both services must be implemented. We introduce a case construct:

```
case $c$ of {
    add ⇒   receive $x$ on $c$
            receive $y$ on $c$
            send $x + y$ on $c$
    neg ⇒   receive $x$ on $c$
            send $(-x)$ on $c$ }
```

The type of the client side uses the dual constructor $\oplus$ (*choice*) to indicate that a choice is made.

$$\overline{S} = \oplus\langle \mathsf{add}\colon !\mathsf{Int}.!\mathsf{Int}.?\mathsf{Int}.\mathsf{End},\ \ \mathsf{neg}\colon !\mathsf{Int}.?\mathsf{Int}.\mathsf{End}\rangle$$

A particular client implementation makes a particular choice, for example:

```
addclient            negclient

select add on c      select neg on c
send 2 on c          send 4 on c
send 3 on c          receive x on c
receive x on c       code
code
```

Note that the type of the subsequent interaction depends on the label which is chosen. In order for typechecking to be decidable, it is essential that the label add or neg appears as a literal name in the program; labels cannot result from computations.

Protocols which specify repetitive behaviour require recursive session types. In a typical protocol, some branches lead to End while others lead to recursive occurrences of a type variable. Alternating nesting of & and $\oplus$ describes complex dialogues, for example when certain requests by the client may produce responsed indicating either successful or unsuccessful execution.

Complete systems can be constructed by using parallel composition: a client and a server in parallel would communicate along a shared channel. It is essential that any session channel is used by just two processes at a time: one at each end. Therefore the type system must track the ownership of the two ends of a session channel.

2.1 Reformulation of Session Types

In the original formulation of session types, session channels were syntactically distinguished from standard pi calculus channels, special constructs were used to create session channels, and session channels could not be sent from one process to another.

We have reformulated session types so that session channels can be created by means of the standard pi calculus new construct, and transmitted in the same way as standard channels. This change provides greater syntactic regularity, and increased expressive power because session channels become first-class values.

Each end of a session channel can only be used by one process at a time. This restriction is similar to the constraint, studied in linear type systems, that each end of a channel can only be used for a single communication. Our version of session types adapts the techniques of linear type systems, allowing session channels to be transmitted (delegation of sessions) and ensuring that a process which delegates a session does not use the channel again.

We have proved that if a system is constructed according to the static typing rules, then at runtime the actual sequence of messages on any channel is one of the sequences allowed by its session type.

2.2 Subtyping

If a server is upgraded, by the addition of new services or the generalization of existing services, then an existing client, although still correct, might not match the type of the new server. We have addressed this problem by defining a notion of subtyping for session types, according to the usual principle that if $S \leqslant T$ then at any point where a channel of type T is expected, it is safe to use a channel of type S instead. For example, if we have

$$\begin{aligned}
S &= \&\langle \mathsf{add}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\mathsf{End},\ \ \mathsf{equal}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End}\rangle \\
T &= \&\langle \mathsf{add}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\mathsf{End},\ \ \mathsf{equal}\colon ?\mathsf{Real}.?\mathsf{Real}.!\mathsf{Bool}.\mathsf{End}\rangle \\
U &= \&\langle \mathsf{add}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\mathsf{End},\ \ \mathsf{equal}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End},\ \ \mathsf{neg}\colon ?\mathsf{Int}.!\mathsf{Int}.\mathsf{End}\rangle
\end{aligned}$$

then in our theory, assuming that $\mathsf{Int} \leqslant \mathsf{Real}$, we have $S \leqslant T$ and $S \leqslant U$. This indicates that a client which has been verified to follow protocol S (typechecks with a channel of type $\overline{S}$) is guaranteed also to follow protocols T and U.

We have a formal proof of safety of the type system including subtyping.

2.3 Bounded Polymorphism

Our theory of subtyping for session types covers many but not all situations in which a client needs to be verified with respect to an upgraded server. An example which is not covered is the upgrade from S to T, where

$$\begin{array}{rcl} S & = & \&\langle \mathsf{add}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\mathsf{End},\ \ \mathsf{equal}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End}\rangle \\ T & = & \&\langle \mathsf{add}\colon ?\mathsf{Real}.?\mathsf{Real}.!\mathsf{Real}.\mathsf{End},\ \ \mathsf{equal}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End}\rangle \end{array}$$

Bounded polymorphism introduces quantified type variables which can be instantiated by any subtype of a specified bound. We have added bounded polymorphism to session types, associating type variables with the branches of the & constructor. This allows us to replace the above types by

$$\begin{array}{rcl} S' & = & \&\langle \mathsf{add}(X \leqslant \mathsf{Int})\colon ?X.?X.!X.\mathsf{End},\ \ \mathsf{equal}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End}\rangle \\ T' & = & \&\langle \mathsf{add}(X \leqslant \mathsf{Real})\colon ?X.?X.!X.\mathsf{End},\ \ \mathsf{equal}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End}\rangle \end{array}$$

and again we have $S' \leqslant T'$. The key point is that we have now specified that the result type of add is the same as the parameter types.

Bounded polymorphism as a natural combination of subtyping and polymorphism has been extensively studied in the lambda calculus. However, despite the fact that subtyping and polymorphism have been studied separately in the pi calculus without session types, our work seems to be the first presentation of any form of bounded polymorphism for pi calculus.

Again we have a formal proof of type safety for this system.

2.4 Automated Proof

Proofs of type soundness are lengthy and tedious, typically involving large structural inductions with case analysis over a number of syntactic constructors. Dr Gay has investigated the use of automatic theorem proving technology with the aim of producing formalized, machine-checked, and maintainable proofs about pi calculus type systems. A linear type system for the pi calculus (without session types) has been formalized in the Isabelle/HOL theorem proving system, with a certain amount of success in modularizing the proofs. However, automated theorem proving is no panacea, and setting up the necessary infrastructure (particularly for linear types) required substantial effort.

This approach does have promise for future work, but the proofs supporting the results on session types have all been done by hand. To do otherwise would have shifted the whole project into an automated theorem proving project, and this was not felt to be appropriate.

3 Project Plan Review

There was major disruption to the project timetable, and subsequent restructuring of the grant, due primarily to the serious illness of the project student, Malcolm Hole. The only significant change to the spending plans was an increase in staff costs because of the need to provide sick pay for Mr Hole.

The goals of the project shifted away from implementation work, which was originally planned, towards theoretical investigation. This was partly because of the amount of effort (greater than anticipated) needed to provide a sound theoretical basis for subtyping in session types, and partly because of the direction in which Mr Hole's interests developed.

The work on automatic theorem proving was not part of the original proposal, but developed in response to the labour-intensive nature of conducting the necessary proofs by hand.

4 Research Impact and Benefits to Society

Session types, and particularly the definition of subtyping developed during this project, have been used by Vallecillo, Ravara and Vasconcelos to add behavioural information to the interfaces of CORBA objects. This is an interesting direction which could become significant if tool support can be developed.

We consider that the main potential beneficiaries of this work are the designers of concurrent programming languages. We have developed in detail the theory of session types with subtyping and bounded polymorphism, in the context of the pi calculus, and argued that this is a useful type system for client-server development. However, we recognise that our results need to be pushed down this avenue of exploitation, and our plans to do this are described in Section 7 below.

5 Explanation of Expenditure

The only deviation from the original plan is that sick leave has resulted in increased staff costs.

6 Dissemination Activities

Papers presenting the results of the project have been presented at international conferences. Malcolm Hole, the project student, will submit a PhD thesis during 2003, and it is intended that Dr Gay and Mr Hole will submit one or more journal papers during 2003. A number of technical reports have been produced.

Dr Gay lectured at an EPSRC MathFIT Instructional Meeting on Types and Semantics for Concurrency, held in July 1998 at Imperial College, London, and presented some of the project's initial results on session types. Malcolm Hole attended the same meeting.

The website `www.dcs.gla.ac.uk/~simon/novel` collects the results associated with the project.

7 Further Research

If session types are to become useful in practical programming, then they must be transferred from the pi calculus to a mainstream programming language or at least reformulated in one of the standard programming language paradigms. During the last year, Dr Gay has been collaborating with Dr António Ravara (Technical University of Lisbon) and Dr Vasco Vasconcelos (University of Lisbon) on a formulation of session types in a language based on simply-typed lambda calculus with side-effecting input/output operations. This is a step towards an object-oriented language with communication channels and session types; such a language would represent significant technology transfer. This collaboration has been funded by a grant from the British Council in Portugal and the Portuguese Council of University Rectors, under the Treaty of Windsor scheme.