Verification tools for multi-core programming

Final Report

Peter Collingbourne

Supervisor: Dr Paul H J Kelly
Second Marker: Dr Susan Eisenbach
Abstract

Programs that perform dyadic communication usually have no guarantee that their communication is correctly typed. If an incorrectly typed value is transmitted or received, programs may remain oblivious to the problem and exhibit errors or crash. We may characterise a dyadic communication using a type construction known as a session type, such that any communication that is constrained by the session type will be correctly typed. We describe a C++ implementation of a component-based language that enforces session typed communication between its components.
I would like to acknowledge my advisor, Dr Paul H J Kelly, who provided me with a lot of support during this project.
Contents

1 Introduction ............................................. 11
  1.1 Contributions .......................................... 13

2 Background ............................................. 15
  2.1 Session Types ........................................... 15
    2.1.1 In the π-Calculus ................................. 15
    2.1.2 In Object-Oriented Programming Languages .......... 16
    2.1.3 The Ninja Language ............................... 18
  2.2 Component Models ...................................... 19
    2.2.1 Singularity and Sing# ............................ 19
    2.2.2 Fractal ........................................... 20
  2.3 Finite State Automata .................................. 21
    2.3.1 Interface Automata .............................. 21
    2.3.2 Choreography .................................... 21
  2.4 Type Inference ......................................... 22
  2.5 Language Embedding .................................... 23
    2.5.1 Lisp Macros ....................................... 23
    2.5.2 C++ Templates .................................. 23
    2.5.3 Template Haskell ................................. 23
  2.6 Program Analysis and Transformation Tools .............. 25
    2.6.1 SUIF ............................................. 25
    2.6.2 ROSE ........................................... 25
    2.6.3 Stratego ......................................... 25
  2.7 Multicore Processors .................................... 26
    2.7.1 Cell ............................................ 26
    2.7.2 Xeon ........................................... 26
  2.8 Conclusion ............................................. 26

3 Design .................................................. 27
  3.1 Sessions and Channels ................................ 27
  3.2 Participants .......................................... 29
  3.3 Initialisation ......................................... 30
  3.4 Type Mutation and Linearity ........................... 30
  3.5 The receive Complex Form ............................. 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.9 Multi-Channel Communication Procedures</td>
<td>90</td>
</tr>
<tr>
<td>10.10 Participant Behavioural Restrictions</td>
<td>91</td>
</tr>
<tr>
<td>10.11 Delegation to Participants</td>
<td>92</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Languages for programming multi-core systems often provide a means for communication between programs running on different cores. However such communication may not always be well formed. Consider the two programs shown in Figure 1.1, running on two separate cores, where \( c \) is a channel of communication between the cores, and \texttt{read} and \texttt{write} are polymorphic communication functions. By tracing the communication that occurs between these two programs, we can see a type mismatch that occurs when the second datum is sent over the channel – namely, that an \texttt{int} is expected where a \texttt{long} has been sent. Should these programs be executed, undefined behaviour would result, especially if \( c \) is used any further in the program. One way of preventing this problem is to introduce unique identification numbers for each type, which are transmitted along with the type with every \texttt{write}, and are compared with the expecting type with every \texttt{read}. However, this solution only provides for runtime type checking. We introduce a compile-time type checking mechanism which would make these programs untypeable.

```
1. int x;
2. long y;
3. c.read(x);
4. y = x + 1;
5. c.write(y);
```

```
1. int x, y;
2. c.write(1);
3. c.read(y);
```

(a) (b)

Figure 1.1: An untyped communication program
CHAPTER 1. INTRODUCTION

Fundamentally, what we would like to do is to annotate \( c \) with a piece of information which states that only an \texttt{int} may be sent over the channel, and furthermore afterwards only an \texttt{int} may be sent in the other direction. The technique of \textit{session typing} allows us to specify this sort of information through the type system of the language. The notion of session typing we give in this chapter is based on the Ninja \cite{11} C++ extension, upon which this work is based. Under session typing, instead of communicating over channels, communication happens over sessions. Sessions are derived from channels using the \textit{invocation} mechanism. When a channel is invoked a procedure is initiated (perhaps on another core) to handle communication over that session, and a session is returned to the invoker. This session inherits the type of its channel. Communication over the session may occur only in accordance with its type (for example, if the type of the session specifies that only an \texttt{int} may immediately be sent, the only valid type of the \texttt{send} function’s parameter is \texttt{int}). When communication over a session occurs, the type of the session is \textit{mutated} in order to reflect the fact that the communication has occurred. The resultant type is known as the \textit{continuation type}.

It is important to note that in a communication system, there exist two viewpoints, that of the first peer and that of the second. So from the communication procedure’s point of view, any values which are sent are received, and any values which are received are sent. Each communication procedure receives a session as an argument, which is used to communicate with its invoker. A simple transformation is applied to the channel type, replacing sends with receives and vice versa, to derive the type of the communication procedure’s argument. The resultant type is referred to as the \textit{dual} of the original type.

Let us define a simple syntax for session types, where \( \text{seq}(a, c) \) represents sequential composition. \( a \) is an action, either \texttt{in}(t) or \texttt{out}(t), \( t \) being a fundamental type. \( c \) is another session type, the continuation type of \( \text{seq}(a, c) \) after \( a \). \texttt{nil} is a session type representing no possible action. Figure 1.2 shows two

```
int x, y;
session s = c.invoke();
s.write(1);
s.read(y);

c(session s) {
  int x, y;
  s.read(x);
  y = x + 1;
  s.write(y);
}
```

(a) (b)

Figure 1.2: A typed communication program
well-typed programs based on our untyped programs. Part (b) is the communi-
cation procedure, and part (a) is a piece of code that uses it. We can see that the
type of $c$ is $\text{seq(out(int), seq(in(int), nil))}$. Let us first consider part (a) of this
program. The session $s$ allocated on line 2 inherits its type from $c$. At line 3,
an int is sent. This is valid because it is the action specified in the session type.
Then $s$’s type mutates to $\text{seq(in(int), nil)}$. At line 4, an int is received (again
valid due to the session type’s action) and the type mutates to $\text{nil}$. For part (b),
the type of $s$ is the dual of $c$’s type, in this case $\text{seq(in(int), seq(out(int), nil))}$.
The sequence of actions and mutations occurs in a similar way as in part (a),
and we are left with $\text{nil}$ as $s$’s type.

1.1 Contributions

• We describe a design for adapting the Ninja language (Section 2.1.3), a
  component-based language based upon the ideas discussed in this chapter,
to the C++ language (Chapter 3).

• We describe how our design has been implemented in the C++ language,
  using templates as a mechanism for enforcing type constraints (Chapter 4).

• We describe how Ninja is extended to include a delegate (callable proce-
dure) model which is fully typeable (Chapter 5).

• We show the expressiveness of our language by describing the implementa-
tion of a multicore publish/subscribe broker and an election protocol
(Chapter 6).

• Our C++ implementation requires that a new variable be declared after
every communication, making use of the old variable an error. Variables
that have this property are described as being linear. We describe how a
linearity check has been implemented using program analysis techniques
(Chapter 7).

• Our C++ implementation requires that each variable declared in this way
have a unique type. Specification of these types is cumbersome and in-
creases maintenance difficulties. We describe the implementation of a
program transformation that uses type inference to determine the correct
type for each of these variables (Chapter 8).

• We evaluate the performance and functionality of the language (Chap-
ter 9).

• We conclude by giving a description of possible future work and outline
how the Ninja language can be extended to implement these (Chapter 10).
Chapter 2

Background

This chapter describes the background and related work for this project. It first describes the theoretical underpinnings of the session typing system we shall use, as well as the Ninja work on which this project is based. As Ninja is a component model, we then look at some alternative component models. We continue by briefly surveying embedded languages, followed by program analysis and transformation tools. Finally we look at some multi-core architectures.

2.1 Session Types

At the most fundamental level, session types are a mechanism for characterising process interactions. Session types have been implemented in several different contexts, including the \( \pi \)-calculus [10] and object-oriented programming languages [7].

2.1.1 In the \( \pi \)-Calculus

The session-typed \( \pi \)-calculus provides a basis for understanding the concept of session types in more practical contexts. We shall concern ourselves with a simplified version of the session typed \( \pi \)-calculus, based on the work of [10]. We define types inductively in the following way.

1. 1 is a type.
2. Any desired user-defined type, such as \texttt{nat} representing natural numbers, is a type.
3. If \( \delta \) is a type, then \( \uparrow \delta \), \( \downarrow \delta \) and \( \bar{\delta} \) are types.
4. If \( \delta_1 \) and \( \delta_2 \) are types, then \( \delta_1 \& \delta_2 \), \( \delta_1; \delta_2 \) and \( \delta_1 \oplus \delta_2 \) are types.
5. If \( \delta_f \), \( \delta_{1...n} \) are types, then \( \delta_f(\delta_1, \ldots, \delta_n) \) is a type.
\( \uparrow \delta \) is the type of a process that receives a value of type \( \delta \).
\( \downarrow \delta \) is the type of a process that sends a value of type \( \delta \).
\( \delta_1 \uplus \delta_2 \) is the type of a process that can be \( \delta_1 \) and \( \delta_2 \).
\( \delta_1 \otimes \delta_2 \) is the type of a process that executes \( \delta_1 \) followed by \( \delta_2 \).
\( \delta_1 \oplus \delta_2 \) is the type of a process that makes a choice between \( \delta_1 \) and \( \delta_2 \).

Let a name be a member of a universe \( U \) which associates types with names. Actions are defined inductively with a type annotation on each action.

- If \( x^\delta \) is a name, \((\uparrow x^\delta)\downarrow \delta \) (send \( x^\delta \)) is an action and \((\downarrow x^\delta)\uparrow \delta \) (receive \( x^\delta \)) is an action.
- If \( V^\delta \) is an action and \( \delta' \) a type, \( \text{inl}(V^\delta)\oplus \delta' \) (choose LHS of interacting process) and \( \text{inr}(V^\delta)\otimes \delta' \) (choose RHS of interacting process) are actions.
- If \( V_1^\delta_1 \) and \( V_2^\delta_2 \) are actions, \((V_1^\delta_1 ; V_2^\delta_2)\delta_1 \otimes \delta_2 \) (sequential composition of \( V_1^\delta_1 \) and \( V_2^\delta_2 \)) and \((V_1^\delta_1 \& V_2^\delta_2)\delta_1 \uplus \delta_2 \) (choice between \( V_1^\delta_1 \) and \( V_2^\delta_2 \)) are actions.
- If \( P \) is a term, \((P)^1 \) (spawning of the term \( P \)) is an action.

We also define terms as follows:

- If \( a^\delta \) is a name and \( V^\delta \) an action, \( a^\delta : V^\delta \) is a term.

The syntax \( V_1^\delta \odot V_2^\tau \leadsto P \) indicates that actions \( V_1^\delta \) and \( V_2^\tau \) interact to give the set of terms \( P \). This is known as reduction of actions (the exact set of rules is elided for brevity; interested readers may consult [10]). The one-step reduction rule over sets of terms is defined as follows (the congruence rules are given in [10]):

\[
(\text{com}) \quad V_1^\delta \odot V_2^\tau \leadsto P \\
(\delta_1, a^\delta : V_1^\delta, \delta_2) \quad \longrightarrow \quad (\delta_1, P, \delta_2)
\]

\[
(\text{struct}) \quad P_1' \equiv P_1 \quad P_1 \longrightarrow P_2 \quad P_2 \equiv P_2' \\
\quad P_1' \longrightarrow P_2'
\]

2.1.2 In Object-Oriented Programming Languages

The definition of session types in terms of the \( \pi \)-calculus may be applied to an object-oriented programming language, using the techniques of [7]. In this language, session types are used to characterise communication between object methods. Types are defined inductively as follows:

- Built-in types such as \textbf{int} are present, as are user-defined object types.
- \textbf{end} is a type.
- If \( t \) and \( u \) are types, \(?t.u \) and \(!t.u \) are types.
- If \( t_1, t_2 \) and \( u \) are types, \(?<t_1,t_2>.u \) and \!<t_1,t_2>.u \) are types.
- If \( t \) and \( u \) are types, \(?<t>\.u \) and \!<t>\.u \) are types.
2.1. SESSION TYPES

**session** BuyProduct =
begin ! String. ? double. !<! Address. ? DeliveryDetails. end, end>

**session** RequestDelivery =
begin ! ProductDetails. !( ? Address. ! DeliveryDetails. end). end

Figure 2.1: Session types for a buyer-seller-shipper example (courtesy of [7])

![Sequence diagram](sequence_diagram.png)

Figure 2.2: Sequence diagram for buyer-seller-shipper example (courtesy of [7])

Figure 2.1 shows two example session type definitions, which may characterise the communication shown in Figure 2.2.

The language is a Java-like object-oriented language, with the additional primitives (where u is a channel, s is a session type and e an expression):

- `spawn { e }`
- `connect u s { e }`
- `u.receive(e)`
- `u.send(e)`
- `u.receiveS(u){ e }`
- `u.sendS(u)`
- `u.receiveIf{ e }{ e }`
- `u.sendIf(e){ e }{ e }`
- `u.receiveWhile{ e }`
- `u.sendWhile(e){ e }`

The `spawn` primitive creates a new thread to evaluate the given expression. The `connect` primitive establishes a link of type s along channel u, and evaluates e. The `receive` and `send` primitives are used to communicate values through the channel. The `receiveS` and `sendS` primitives are used to communicate pre-established sessions through the channel. `receiveIf` and `sendIf` are used to implement choice and correspond to the ![t1, t2]>_{u} and ![t1, t2]>_{u} types – the value of the `sendIf` boolean expression is used to decide which of the branches to follow in both the `sendIf` and corresponding `receiveIf`. `receiveWhile` and `sendWhile` are used to implement iteration and correspond to the ![t]>_{u} and ![t]>_{u} types – the loop continues for both parties until the `sendWhile` boolean expression evaluates to false.

Two peers may communicate on the same channel if the peers have dual session types. The dual of a session type is obtained by replacing ? by ! and vice versa (except for any session type sent via the channel).

The analogies between the language and the session typed π calculus are evident. The set of object methods that are waiting for an interaction via the
typedef session sessionex {
    in request(int);
    out reply(int);
    in {
        ok(); end | again(); sessionex
    }
} Sessionex;

Figure 2.3: A Ninja session type

connect primitive may be regarded as a set of terms that may interact via the
one-step reduction rule. The connect primitive itself is analogous to a term of
the form \( u^* : e^* \). The spawn primitive is analogous to an action \( (e)^1 \). receive
and receiveS are analogous to \( (e^t(e))^{t(e)} \), where \( t(e) \) is such that \( e^t(e) \in U \).
Similarly send and sendS are analogous to \( (e^t(e))^{t(e)} \). sendIf is analogous
to inl(e1) or inr(e2) depending on the value of the conditional, and receiveIf
is analogous to e1 & e2.

2.1.3 The Ninja Language

The Ninja language [11] is a C++-like language based on the concept of session
types. Ninja is designed for use in multicore applications. An example of a Ninja
session type is shown in Figure 2.3. From this example we can see that the ;
operator indicates sequential composition, as with the session-typed \( \pi \)-calculus.
The | operator is similar to & and \( \oplus \) in the session-typed \( \pi \)-calculus, except that
Ninja allows for more than two choices at once.

The building blocks of a Ninja program are participants. Each participant is
associated with one or more channels, each of which contains an associated session
type. Communication between participants takes place by invoking these
channels to obtain a session. There are two types of channel: linear and shared.
Linear channels can only be invoked once, whereas shared channels can be in-
voked an unbounded number of times. In general, one participant invokes the
channel (receiving a session whose type is the dual of the session type of the
channel) and the other defines a procedure for communication over the channel.
The process of invocation may be considered analogous to the \( \pi \)-calculus action
\( s: C \), where \( C \) is a continuation of the procedure past the point of invocation
and \( s \) is a symbol associated with the communication procedure by means of
a term of the form \( s : P \) (for linear channels) or \( !s : P \) (for shared channels),
where \( P \) is the body of the procedure.

As previously mentioned, Ninja is based on the C++ language. Ninja adds
the following communication primitives to the language: invoke, send and
receive. The invoke primitive carries out the invocation operation as described
above. The send primitive takes two arguments; the session and the datum
to be sent. This primitive will send its second argument along the session
2.2. COMPONENT MODELS

Figure 2.4: An example of the Ninja receive complex form

specified in the first argument, and alter the session’s type to reflect the fact that communication has occurred.

The receive primitive has two forms: a simple form which takes one argument (the session) and returns a value of the type that can be immediately received according to the session type, and a complex form which can distinguish between multiple distinct types which may have been received at this point. The complex form is structured like a switch statement, where the different ‘cases’ are the different types which may be received. An example of the complex form is shown in Figure 2.4. Note that both of these forms will also alter the session type to reflect the communication, and for the case of the complex form the resultant session type will depend on the type of the datum that was received.

For initialising participants and channels, Ninja offers two primitives: newchannel and spawn. The newchannel primitive creates a new channel and takes two arguments: the type of the channel (which is defined in terms of the type of the sessions that the channel will produce once invoked) and the core on which the channel’s buffer will reside (normally the core on which the communication procedure for that channel will run). The spawn primitive spawns a participant on a particular core. It takes two arguments: the participant (along with the channels it shall use) and the core.

Ninja may be considered a component model since it defines an architecture for its components (the participants) to interact in a particular manner.

2.2 Component Models

We shall now examine a selection of component models, such that they may be contrasted with the Ninja component model.

2.2.1 Singularity and Sing#

Singularity [12] is an experimental operating system which has been designed with the primary goal of dependability. Singularity is primarily written in an experimental variant of the C# language known as Sing#, which allows for pre- and postconditions, invariants and built-in communications primitives.

In Singularity’s component model, a component is known as a SIP, or Software-Isolated Process. A SIP may be a device driver, system process, application or application extension (however these are not the only possibilities). Each SIP communicates with others over communication channels, which are constrained
by a channel contract (the parallel to the session types of the \( \pi \)-calculus), and an example is shown in Figure 2.5.

We can see some obvious similarities between a channel contract and a session type. \(?\) and \(!\) are used to denote incoming and outgoing messages respectively. The \(\rightarrow\) connective performs the same function as the \(;\) connective, and \(\lor\) is analogous to \& or \(\oplus\) (depending on context). However there are some significant differences. The most important difference is that although the communication sequence is given in the type definition, it is not actually a part of the type system itself. This implies that any verification must take place outside of the type system. Channel contracts are enforced by a combination of static verification and runtime monitoring, however the state machine checks are currently runtime only.

An innovative technique used by Singularity is its approach to interprocess garbage collection. The language has been designed so that garbage collection may occur on a per-process basis. This means that no two processes may hold a reference to one particular object at once. This would cause problems when the communication mechanism is employed to send objects through a channel, as it would normally mean that both processes would hold a reference to the object. Singularity solves this problem by deleting the (only) reference to the object in the sending process when the message is sent. This constraint is enforced by the language itself (more details are found in [12]). The use of this technique paves the way for optimisations such as zero-copy message passing to be implemented (aliasing issues are eliminated by the aforementioned constraint).

### 2.2.2 Fractal

Fractal [4] is a flexible language-independent component model which is designed to reduce development costs. It has seen many uses, including in operating system design [20] and graphical user interfaces.

Central to the Fractal design philosophy is the interface. An interface is essentially a list of procedures that characterises interaction with a component, similar to Java interfaces in a way. The definition of a Fractal component consists of one server interface and any number (including zero) of client interfaces.
A component exposes its functionality via the server interface and may invoke other components using its client interfaces. Components are composed by connecting (or binding) a client interface to a compatible server interface.

In order to provide encapsulation, a Fractal component may contain other components (known as sub components). In this case, sub components may be connected within the component in any way desired. For such a component to provide services to the outside world, a sub component’s server interface may be made available via an export binding and client interfaces via import bindings.

In particular, in the operating system component architecture [20], the Fractal component model is used to allow for components to be replaced via dynamic reconfiguration or ‘hot-swapping’. Cited advantages of this approach include seamless system upgrades, adaptation of a system to changing conditions (by substituting in components that are better suited to the current situation) and dynamically insertable instrumentation capabilities. Examples of operating system components include device drivers, memory allocators and schedulers.

2.3 Finite State Automata

By contrast with the session type approach to characterisation of communication mechanisms, we shall now examine another popular tool – finite state automata – which has been adopted by a number of authors for the same purpose.

2.3.1 Interface Automata

Interface automata are introduced in [6], and is based on separability of a software system into components which communicate via messages. Each component consists of a finite-state automaton and a number of inputs and outputs. Inputs trigger the relevant labelled transitions in the automaton, whereas outputs are triggered by the available transitions. The mapping between components’ inputs and outputs is generally defined by the programmer.

The authors define a composition operator $\otimes$ which composes two automata such that any corresponding input and output actions occur simultaneously. If an input action can occur in the composed automata such that no output action may occur, or vice versa, the automata will enter an illegal state. Therefore correctness is ensured by composing the automata that comprise a software system and verifying that no transitions to illegal states exist.

2.3.2 Choreography

[8] presents an alternate approach to using finite state automata to verify correctness, and applies it to the Web services environment. The authors propose a tool, LTSA-BPEL4WS, which can translate a Web service specification in BPEL4WS, an XML-based language for modelling the behaviour of a web service, to a finite-state automaton in the Finite-State Process (FSP) language, which can be compiled using the Labelled Transition System Analyser (LTSA).
CHAPTER 2. BACKGROUND

t apply(f:t> x) {
   ... 
}

int run(x1) {
   x2 = apply(x1);
   nil x3 = apply(x2);
}

Figure 2.6: Simple pseudocode example of type inference by unification

\[\begin{align*}
\sigma_{x_1} &= f(\sigma_{x_2}) \\
\sigma_{x_2} &= f(\sigma_{x_3}) \\
\sigma_{x_3} &= \text{nil}
\end{align*}\]

Figure 2.7: The set of constraints derived from Figure 2.6

Since Web services may be composed, or choreographed, composing the components in a Web service allows us to check for safety and liveness properties, as the authors describe.

As opposed to the interface automata approach in which the automaton is the program itself, BPEL4WS is a modelling language in which specifications must be written manually. The advantages of manual modelling are language independence and less complexity, however there is a risk of false positive verification results due to incorrectly specifying the model.

2.4 Type Inference

Type inference is a technique for reducing the verbosity of a language by making its type declarations implicit. Type inference can also be used to deduce properties about a program. A number of techniques are present in the literature for type inference, the most prolific being that of Milner [16]. Milner’s technique is based on the idea of unification. Although Milner applies his technique to functional languages, the same technique can be used for procedural languages since we only need to be able to produce a set of constraints for the unification to proceed. By way of example, Figure 2.6 is a simple program in

\[\begin{align*}
\sigma_{x_1} &= f(f(\text{nil})) \\
\sigma_{x_2} &= f(\text{nil}) \\
\sigma_{x_3} &= \text{nil}
\end{align*}\]

Figure 2.8: The solution to the set of constraints in Figure 2.7
a procedural language, where we would like to infer the type of the variable x1. Figure 2.7 shows the set of constraints and Figure 2.8 shows the solution to these constraints.

2.5 Language Embedding

Certain languages allow for the embedding of a domain-specific language that conforms to a particular programming model. [5] surveys three programming languages that support domain-specific languages.

2.5.1 Lisp Macros

Common Lisp has provided facilities for domain-specific languages for over 20 years. The prototypical example is the Common Lisp Object System [3], a complete object-oriented extension to Lisp implemented entirely within the language. Domain specific languages are implemented via macros, which are similar to functions except that instead of returning a value, an abstract syntax tree is returned. Since in Lisp, executable programs (known as forms) are a subset of data, one can easily create the AST using the normal data manipulation facilities provided by the language.

2.5.2 C++ Templates

The C++ [25] template mechanism is a powerful system for generating specialised code at compile time. [26] showed that C++ templates are Turing complete, and [24] demonstrated how the language’s type system can be extended using templates in order to validate documents according to a XML schema.

C++ templates’ usefulness result from the ability to combine expressibility (the template system is essentially a compile-time functional language embedded within C++) and functionality (despite the apparent limited syntax, templates can parameterise almost any construct in C++). A powerful C++ technique result from the ability to parameterise an object’s superclass. For example Figure 2.9 is a composition template that inherits from all classes in the given list and calls their run methods sequentially when its own run method is called. Due to the ability of many compilers to perform inlining, this code turns out to be as efficient as calling the run methods individually.

2.5.3 Template Haskell

Template Haskell [23] is an extension to Haskell that is more flexible than C++’s template system in that it allows generation of function definitions at compile time by creating an abstract syntax tree. In this way it is similar to Lisp macros, but because Haskell is a strictly typed language, type checking is performed when a template is expanded. Similarly to Lisp, there are quoting ([[1 1]]) and
struct nil {};  

template <typename Car, typename Cdr = nil> struct cons {
    typedef Car car; typedef Cdr cdr;
};

template <typename L>
struct comp : L::car, comp<typename L::cdr> {
    void run() {
        L::car::run();
        comp<typename L::cdr>::run();
    }
};

template <>
struct comp<nil> {
    void run() {}
};

Figure 2.9: Generic object composition in C++

comp n = do {
    x <- qNewName "x" ;
    comp' n (varE x) x
}

where
    comp' 0 a x = lamE [tupP [varP x]] a
    comp' n a x = do {
        f <- qNewName "f" ;  
        lamE [tupP [varP f]] (comp' (n-1) (appE (varE f) a) x)
    }

Figure 2.10: Generic function composition in Template Haskell
unquoting ($$) operators. Because of the relative (to Lisp) complexity of the AST, construction functions can be used in place of the quotation operator. Figure 2.10 shows how to create a function that will take as input $n$ functions and return a composed version of them.

2.6 Program Analysis and Transformation Tools

Such tools are designed to operate on either a source-level or a bytecode representation of a program. An internal representation is generated, which can be manipulated by the user or by any number of predefined transformation passes. After manipulation, the internal representation can be output in such a form that it can be used to generate an executable binary. In this way, programmatic transformations of source code are possible.

2.6.1 SUIF

The Stanford University Intermediate Format [30] is structured as a kernel, which defined the underlying representation, on top of which is a toolkit, which provides various analyses and optimisations. It contains C and Fortran front ends, but is potentially flexible enough to handle any language. SUIF can build binaries using an optimised MIPS backend, as well as C source code, which in turn can be used to build binaries using a conventional compiler.

2.6.2 ROSE

ROSE [22] is a tool for building source-to-source translators. Its Intermediate Representation (IR) can represent C, C99 and C++ codes, with Fortran support in development. ROSE was initially conceived as an optimisation tool for specialisable numerical libraries such as Overture, but has expanded in scope to be used in many fields including automatic loop parallelisation [21] and alias analysis [29].

ROSE consists of a frontend, midend and backend. The frontend parses a source-level representation of the program using a commercially available frontend. The midend manipulates the IR and the backend performs the reverse operation to the frontend in that it ‘unparses’ the IR into source code, which can be used to build binaries with a conventional compiler.

The crux of ROSE’s usefulness is that is one of the only tools of its kind that can operate on C++ codes. This is because C++ is inherently hard to parse [13] and has many details that must be handled correctly (the fact that the specification [25] contains over 900 pages should be some hint to this). ROSE partially sidesteps this issue by using the commercially-available EDG frontend.

2.6.3 Stratego

Stratego [27] is a specification language for a term rewriting system. It is based on the rewriting strategies paradigm. An example of a rewriting strategy would
be a top-down traversal, however there are more sophisticated traversals available.

Stratego’s internal representation is well specified, and is known as the Annotated Term Format, or ATerm. Interfaces for common languages such as Java are available, that translate code between the source language and ATerm.

During a traversal, nodes are matched and rewritten against a set of rewriting rules in System S, a language that allows for definitions of tree transformations.

2.7 Multicore Processors

In the wake of the increasing costs of feature size reduction, and the possibility of coming out of step with Moore’s Law, chip designers have adopted the technique of placing multiple processing cores on a single chip.

2.7.1 Cell

The IBM/Sony/Toshiba Cell processor \cite{15} is a multicore processor based on one core based on the POWER architecture (the PPE) and eight custom SIMD cores (the SPEs). The design has been tailored for applications such as computer gaming (Sony’s latest video game console, the PlayStation 3, uses a Cell core with 7 SPEs) and distributed computing \cite{1}.

The cores are connected with a coherent Element Interconnect Bus (EIB), and data is sent between the cores using DMA commands. SPEs also present a mailbox which can be used as a FIFO for communication between cores.

2.7.2 Xeon

Xeon is a line of Intel chips designed for server platforms. The most recent Xeon chips are dual- and quad-core implementations of the x86-64 architecture. As opposed to the Cell and other specialty processors, Xeons implement a standard PC architecture. Any communication features the chip may have are only exposed to the operating system.

2.8 Conclusion

We have looked at a number of process algebras, languages, tools and processors which shall aid us in our implementation work.
Chapter 3

Design

This chapter shall describe how the Ninja language has been adapted to standard C++, without the use of any special compilers or language extensions. This adaptation is impure in that the typing system is sound only under a rule known as the linearity constraint which cannot be enforced in pure C++. We will see how we can implement a program analysis to verify that the linearity constraint holds for a given program in Chapter 7.

This section shall only describe the user-visible portions of the adaptation; the implementation details shall be covered in Chapter 4, Implementation.

3.1 Sessions and Channels

The most important decision we must make is that of how to represent session types. The easiest way would be some form of string representation embedded within the session, however this would not afford us some of the expressive power we shall see later. A session type is fundamentally a strictly specified hierarchy of types, and the C++ template mechanism allows us to specify a user-defined type hierarchy. So we shall use templates as our representation mechanism.

We distinguish between Ninja actions and sessions. An action is a primitive communication step, such as in int, whereas a session is a fully specified session which may include sequential composition, choice etc. Actions in our language 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 an arbitrary number of sessions, so we have a choice of a tuple-style representation of the form choice<S1,S2,...,Sn> or a linked-list style representation of the form choice<S1,choice<S2,...,choice<Sn>..>>. Although the former representation limits the number of choices we are allowed
struct s;
typedef seq<in<int>,call<s>> r;
struct s { typedef r t; };
3.2. PARTICIPANTS

use the session types directly, but this would require that each possible individual session type (i.e. \texttt{seq}, \texttt{choice}) implement the methods \texttt{send} and \texttt{receive}. This is non-ideal, as it may involve excessive code duplication for each new session type we introduce, where we may be able to consolidate these concerns. Instead we introduce a \texttt{session\langle S\rangle} type for sessions, where \texttt{S} is the session type. Similarly we have \texttt{channel\langle S\rangle} for channels.

3.2 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 communications 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;

- a \texttt{spawn} method which spawns the participant.

There are two implementation alternatives. The first is an unparameterised \texttt{participant} class, which is provided with three static arrays of linear, shared and invokable channels. The second is a \texttt{participant} class which is parameterised over its channel types. Both would implement the constructor and \texttt{spawn} method. For the first alternative we would be unable to verify the correctness of the types of the parameters, which would lead to runtime errors should an incorrectly typed channel be given. Thus we must use a parameterised \texttt{participant} class.

The \texttt{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 \texttt{os::default\_location}.

How should we supply parameters to the \texttt{participant} class? The obvious strategy for encapsulating a participant’s communication procedures and channel variables is to make them non-static members of their own class, known as the participant implementation class. Now we must decide upon the relation between the participant implementation class and the (parameterised) participant class. There are three strategies we must consider:
1. Implementation class is a subclass of participant class
2. Participant class is a subclass of implementation class
3. Participant class contains implementation class (i.e. the participant class possesses a field of the implementation class type)

We can discard strategy 1 immediately. 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. This leaves strategies 2 and 3. There is no practical difference between these strategies (except for restrictions on naming, visibility etc.) so in the end strategy 2 was chosen.

The participant’s template parameters will thus comprise its base class (the implementation class) and the list of channels. Ideally each channel definition would take the form channel_type<s, p1, p2, ...> where s is the channel’s session type and p1, p2, ... are parameters specific to the channel type, such as the name of the communication procedure. However the definition of the type of the parameters pk is likely to depend on two factors: the session type, and the participant base class (this is especially true for cases where the parameter comprises a class member reference). The channel type template does not know which base class is being used (as it is not one of its parameters), but the participant does. So these parameters cannot be parameters to the channel type, and we are forced to make them participant parameters (this restricts the number of parameters per channel). Now the parameter type can incorporate base class information as well as channel type-specific information. Details of how this is done are given in chapter 4, Implementation.

An example of a skeleton participant is shown in Figure 3.2.

3.3 Initialisation

There must be a section of the program that performs initialisation. Initialisation would entail instantiating channels as well as participants connected to these channels. Effectively the participant black boxes are connected using channels in the initialisation phase. Where should this initialisation happen? The C runtime library for any platform will start the main function when an executable is executed. So programs will perform their initialisation in the main function, or a callee thereof.

3.4 Type Mutation and Linearity

The Ninja language specification states that after a session has performed a receive or send operation, its type must automatically mutate to the session’s continuation type relative to the action that has taken place. The C++ language does not permit a variable’s type to mutate under any circumstances,
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;

Figure 3.2: An example of a skeleton participant

although it does 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 regrettably switch to the strategy of introducing a new session variable after
each communication action.

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 [19] 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 ini-
tialiser. 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.

It is easy to imagine how verbose each intermediate session variable decla-
ration can become, especially if the program is using a particularly long, cum-
bbersome session type. This verbosity may lead to hard-to-trace compiler errors
and maintenance difficulties. To partially alleviate this problem, a syntax is
introduced to make intermediate session type specifications as short as possible.
The type c_session<S, A1, A2, ..., An>::t is equivalent to the type S after the
actions A1 through An are applied to it (the ::t part is necessary as c_session is
a function over types; functions are introduced in chapter 4, Implementation).

Even if we derive the full type at each stage instead of giving the resultant
type, we still need to give each individual action that has happened so far in the
type specification, which may still result in maintenance difficulties (suppose
the set of actions that occur before a block of code changes). We shall see a way
of circumventing this problem using type inference techniques in Chapter 8.

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 a linear variable, and any program that satisfies this property is said to satisfy the linearity constraint. It would be trivial to implement a runtime linearity check – an internal flag is set by the session when it is used, and any use of the session with the flag set would trigger a runtime error. However this solution is suboptimal as we can determine statically whether a program satisfies the linearity constraint using program analysis techniques. Details of how this has been achieved are given in Chapter 7.

3.5 The receive Complex Form

How may we map the Ninja receive complex form onto our language? First of all we must give a mapping for the ‘switch’ statement; this is implemented as a macro and takes the session type (for implementation reasons) and session as parameters. Following the ‘switch’ statement is a block containing ‘case’ statements. The ‘case’ statement for each incoming type must store the resultant session somewhere. There are two alternatives: declare a session variable for this purpose, or store the session in a pre-existing variable. In both cases, the relevant macro (session_choice_case and session_choice_case_nodecl respectively) takes the destination session, the variable to receive the datum into and a block that handles the case. Any block of session_choice_cases must end in a session_choice_end, again for implementation reasons.

Putting it all together, Figure 3.3 is a translation of Figure 2.4 into our language.

```c
session_choice (stype, s) {
    session_choice_case_nodecl (s1, ok()) {
        break;
    }
    session_choice_case_nodecl (s2, again()) {
        continue;
    }
    session_choice_end;
}
```

Figure 3.3: Translation of Figure 2.4 into our language
3.6 Algebraic Types

The Ninja specification gives a syntax for specifying algebraic types. An algebraic type is a named tuple of basic data types of any size, including zero. In Ninja the algebraic type definition is given in the session type itself. However in C++ it is impossible to give type definitions within a template instantiation (which is all that a session type is). So we must define algebraic types separately.

One might think that algebraic types would be very simple to define – it ought to be possible to define a template whose instantiations are algebraic types. However it is not as simple as this. We must be able to distinguish algebraic types in order to assign them different identifiers, for reasons described in Chapter 4. However there may be multiple algebraic types which take the exact same tuple of parameters. Since these types’ templates will take the same parameters, the types will not appear distinct. As algebraic types may accept zero parameters, the problem is especially severe. So instead of using templates for this, we define a macro DECLARE_ALG_TYPE which expands to the definition of an algebraic type. This solution works because classes with the exact same definition will appear to the compiler to be distinct. The macro shall take the following parameters: the type identifier (see Section 4.2 for more information), the name of the algebraic type, a tuple of its parameter types and the number of parameter types (for implementation reasons).

3.7 Dynamic Channels

A dynamic channel is a channel type which is new to our implementation of the Ninja language. It is a channel which may be invoked any number of times but which is handled by a single instance of the communication procedure. The dynamic aspect of the channel refers to the list of participants connected to the channel, which can change at any time.

In order to properly introduce dynamic channels, we must first introduce session handles and session lists. A session handle is a ‘permanent’ session. It is permanent in the sense that it does not possess the linearity property. Session handles are designed for recursive sessions which return to their original type after being used. The use of a session handle is based on a checkin/checkout principle. If a program wishes to use the underlying session in a session handle, it first checks it out. After the program has used it, and it has returned to its original type, the session is checked in to the handle. While a session is checked out, a mutex is held on the handle. This ensures that no other thread can gain access to the underlying session, which would violate the linearity constraint.

A session list is essentially a list of session handles. It also provides us with some features which cannot be implemented using standard sessions. The most important feature is the ability to listen on multiple sessions at once. The choose_session primitive takes three parameters – the session type (required for implementation reasons), the name for a session handle and a session list – and a block of code. When we receive input on a session, control is relinquished
to the block of code. The block has available to it a handle to the session that has received input under the name of the session handle that was supplied to choose_session. Typically the participant will check the session out, perform communication over it and check it back in.

The main procedure that is used by a dynamic channel is the communication procedure. The procedure is supplied with a session list containing all of the sessions that are currently active on the channel. The key is that the list is maintained automatically by the participant, so for example if the channel is invoked by another participant, the participant’s session is automatically added to the list. A typical communication procedure will consist only of a call to the choose_session primitive and its corresponding code block.

Sometimes we do not wish to store sessions in our session list in the same type as that in which they arrive. We may first wish to initialise the session by performing communication over it, or perform some other bookkeeping tasks. For this purpose an initialisation procedure is provided as an extra parameter. The initialisation procedure will convert a session of the initial type into a session handle of the stored type. The initial type is that which is visible to channel users; it is the type of the channel when it is invoked. The stored type is the session type of sessions in the session list.

3.8 Session Forking

A session fork is a session type which is new to our implementation of the Ninja language. Sometimes we wish to be able to communicate over a channel using multiple sessions at once. We could achieve the same effect using multiple channels, but there would be no practical way to associate one session with another. A fork session type is of the form \texttt{forked<} t_1,t_2,\ldots,t_n\texttt{>}. In order to obtain sessions of types \texttt{t_1}\ldots\texttt{t_n} from a session of this type, we use the \texttt{fork} primitive. In this case, \texttt{fork} is a session method. To obtain the sessions \texttt{s_1}\ldots\texttt{s_n} of types \texttt{t_1}\ldots\texttt{t_n} from a forked session \texttt{s}, the \texttt{fork} method is used as follows:

\begin{verbatim}
s.fork(s1,s2,...,sn);
\end{verbatim}

Session forks are useful in implementing channels which may have both incoming and outgoing activity at the same time. Consider a session of type \texttt{choice<seq<in<} \texttt{int}>s1\texttt{>,seq<out<} \texttt{int}>s2\texttt{>}> used for communication between participants A and B (B uses the dual). A sends a message of type \texttt{int}. Now A believes that the session is of the type \texttt{s2}. Simultaneously, B also sends a message of type \texttt{int}. Now B believes the session is of the dual of type \texttt{s1}. We have now lost our type correctness property.

To work around this problem, we simply fork the session into an incoming and an outgoing session. Data can now flow over these sessions without risk of desynchronisation. We can also implement an optimisation for this special case where one session is used exclusively for incoming data and one exclusively for outgoing data. In this case we can use the same physical communication channel for both sessions. This is possible because the incoming and outgoing data will not interfere with each other.
3.9 Conclusion

We have described the design of our adaptation of Ninja to C++, as well as our own constructs which are new to our implementation. These constructs, when put together, form a fairly complete session typed language embedded within C++.
Chapter 4

Implementation

This chapter shall describe how the Ninja language has been implemented, both with regards to its implementation in the C++ language and to its concrete message-passing implementation for the Unix family of operating systems.

4.1 Functional Template Metaprogramming in C++

The C++ compile-time functional language is one over types and scalar values, but its power is derived from being over types. A function is a template, its arity the arity of its parameters, and its result the type defined by a typedef contained within the template’s result, a struct. Our technique for functional metaprogramming using the template system is based on the ability to perform pattern matching on parameterised type structures (a.k.a. templates).

A template may be instantiated with fewer actual parameters than there are formal parameters, provided that the remainder of the formal parameters have default values. We can exploit this feature of the language by performing computations (calling functions) within ‘unseen’ formal parameters by performing the function calls within their default values, and using the result of these computations during pattern matching.

A template parameter may be a template itself. This allows us to create higher-order functions over types.

Recall that the result of parameterising a template is a struct. Thus we are not restricted to a single type result from this parameterisation. Indeed, we may return multiple type results, or even functions themselves (since templated structs may include other templates). By constructing a set of nested templates, each template accepting a limited number of parameters, we can use all the parameters within the most nested template. Thus we are able to simulate the functional language technique of currying.
4.2 Type Identification

We use the technique of [9] to assign an identifier to each type. Although the identifier assignment is only a side-effect of the author’s main intention of producing a portable `typeof` operator for C++, we are still free to use the identifiers however we choose.

The technique provides us with a MSG_TYPEID unary function, which extracts the type identifier corresponding to the type of the given expression, and a `typeof` unary function which returns the type of the given expression.

This technique requires us to register each type that we use with the above primitives using the DECLARE_MSG_TYPE macro which takes two parameters: the type identifier and the type. Because every type must be registered, it would be infeasible to use this technique for extracting the type of a session variable, however we still find use for it when we need to extract primitive and algebraic types. In fact, all the primitive integral types are preregistered with our library.

4.3 Wire Protocol

For those cases where a session-based stream is used for communication (for example, in the UNIX implementation), the following wire protocol is defined.

For maximum efficiency, the data that is sent over the wire is the physical memory representation of the object that is being sent, according to the host system. This means that factors such as endianness and padding are defined by the architecture/OS/compiler combination. For cases where multiple data types may be sent (i.e. if the immediate session is of the choice type), a type identifier is transmitted before the data. This type identifier is a 32-bit integer which is unique to each type. The type identifier is derived using the technique shown in Section 4.2.

4.4 Sessions and Channels

Both sessions and actions are functions which provide two results: a base type and a choice base type. These base types provide a class for the session class to inherit from. Each base type must provide a do_receive function for each type that may be immediately sent across the session, and a do_send function for each type that may be immediately received. An action will provide a do_receive (for in actions) or a do_send (for out actions) which takes one parameter of the action’s data type. The seq function will inherit the base type and choice type from its first argument, the action.

The distinction between the base type and the choice base type is that for a choice base type, the do_receive and do_send functions respectively verify and transmit the type identifier that corresponds to the type that is being received or sent. This distinction is necessary because only in the case where the immediate session type is a choice do we wish to use type identifiers. Thus the base type of an action provides a do_receive or do_send which does not do this, and the
4.4. SESSIONS AND CHANNELS

\[
\text{ct}^+(a_1, \text{seq}(a_2, s)) = \begin{cases} 
\text{rc}(s) & \text{if } a_1 = a_2, \\
\bot & \text{otherwise}
\end{cases}
\]

\[
\text{ct}^+(a, \text{choice}(s_1, s_2, \ldots, s_n)) = \max(\text{ct}^+(a, s_1), \max(\text{ct}^+(a, s_2), \ldots, \\
\max(\text{ct}^+(a, s_{n-1}), \text{ct}^+(a, s_n)))
\]

\[
\text{rc}(s) = \begin{cases} 
n\text{unroll}(c) & \text{if } s = \text{call}(c), \\
s & \text{otherwise}
\end{cases}
\]

\[
\max(\bot, \bot) = \bot \\
\max(s, \bot) = s \\
\max(\bot, s) = s \\
\text{ct}(a, s) = \text{ct}^+(a, s) \text{ if } \text{ct}^+(a, s) \neq \bot
\]

Figure 4.1: The \textit{continuation_type} function (ct here).

choice base type provides \texttt{do\_receive} or \texttt{do\_send} which does. Furthermore the resultant base type and choice base type of the \texttt{choice} function are both the \textit{union} of the choice base types of their parameters. For our purposes the union type of a set of types is a class which inherits from all of these types, but has no properties of its own.

The \texttt{session} class has as its base class the base class of its parameter, the session. This class is responsible for implementing polymorphic \texttt{receive} and \texttt{send} functions which call the \texttt{do\_receive} and \texttt{do\_send} functions from the correct base class. One may think that it should be possible to call the \texttt{do\_receive} and \texttt{do\_send} functions directly from the \texttt{session} class, due to overloading. However in C++ overloading does not work when the functions with the same name come from different classes, which is the case here. Instead the name of the correct base class must be specified in the function call. This poses a problem for us as we do not know the name of the base class, since it is an implementation detail of the session type (even if we hardcode the name of the class into the \texttt{receive} and \texttt{send} methods, the base type and choice base type classes are bound to be different). Instead we use two functions, one for \texttt{receive} and the other for \texttt{send}, which takes two parameters, the session and the data type, and returns the correct base type.

The concept of the continuation type is implemented in the \texttt{continuation\_type} function. It takes two parameters, a session and an action, and returns the type of the session after the action. If the action cannot be applied to the particular session, the result is undefined (this is represented by an empty \texttt{struct} with no type field). If the result is undefined, any attempt to use the result of the function will render the user untypeable and thus uncompileable. This technique is used to great effect in \texttt{session’s receive} and \texttt{send} methods. If the action cannot be applied to the session, the particular instantiation of the \texttt{receive} or \texttt{send} method will be made untypeable. In this way we restrict the parameter types that can be used with \texttt{receive} and \texttt{send}.

The \texttt{continuation\_type} function function is implemented functionally using pattern
CHAPTER 4. IMPLEMENTATION

matching. A pure functional description is given in Figure 4.1, which is easily translatable into C++. Note in particular the use of the unroll function. This performs the isorecursive unroll operation on the given type (in reality it returns the type contained in the t field of the type).

4.5 Participants

As we have mentioned in Chapter 3, each channel type takes some parameters whose type is solely defined by the type of the channel and types of parameters that have come before it, and some parameters whose type is defined by a combination of the channel type and the name of the implementation class. Parameters of the first type are parameters to the channel type, and parameters of the second type are parameters to the participant. We can say that a channel type is also a function because its instantiation provides to us results in the form of types and functions. In order to allow for two parameters of the second type, two of the results that the channel type function provides – function_type and extra_type – are functions themselves. Given a single parameter (the type of the implementation class), the functions both provide a single result; the types of the two parameters. These types are then used by the participant template declaration to define the types of these two parameters.

After we have obtained the parameter values, we use another function provided by the channel type, chan_decl_type. It takes three parameters: the implementation class and both parameters provided to the participant instance. The result it provides is the type of an action class which is responsible for initialising and running the channel. The participant will hold an instance of the appropriate action class for every channel. When the participant is spawned, the action classes’ channel_runner methods are called. This method should perform any necessary initialisation, and start any threads which are required for running the channel. Threads are spawned using the os::start_thread procedure, which is a platform-independent procedure provided for this purpose (note that at the moment, only a UNIX implementation exists).

4.6 Algebraic Types

The macro DECLARE_ALG_TYPE has been implemented to produce the definition of an algebraic type as well as any other definitions that are required to allow our implementation to recognise the given type as an algebraic type. The macro is implemented using the Boost preprocessor library in order to save implementation time. It would not be a worthwhile exercise to examine the definition of the macro itself simply because it is irrelevant for the current discussion, but for illustration purposes the expansion of the macro DECLARE_ALG_TYPE(32, Type1, (int, long), 2) is shown in Figure 4.2.

From this expansion, we can identify three main code fragments:

1. The definition of the algebraic type;
4.6. ALGEBRAIC TYPES

```cpp
struct Type1 {
    typedef int T1;
    typedef long T2;

    T1 *o1;
    T2 *o2;
    const T1 *co1;
    const T2 *co2;

    Type1(T1 &o1, T2 &o2) : co1(&o1), co2(&o2), o1(&o1), o2(&o2) {}
    Type1(const T1 &o1, const T2 &o2) : co1(&o1), co2(&o2) {}

    void read(os::session &sess) const {
        session_reader<T1>::read(*o1);
        session_reader<T2>::read(*o2);
    }

    void write(os::session &sess) const {
        session_reader<T1>::write(*co1);
        session_reader<T2>::write(*co2);
    }
};

template <>
struct is_alg_type<Type1> {
    static const bool v = true;
};

DECLARE_MSG_TYPE(32, Type1)
```

Figure 4.2: Sample expansion of the DECLARE_ALG_TYPE macro
2. A specialisation of the \texttt{is\_alg\_type} function to return \texttt{true} for this type;

3. A declaration of the type's identifier (required for all types that are sent across a session).

The first thing you should notice about the type definition itself is that there are const and non-const versions of the properties and constructors. The reason for this is that there are two ways we can use an algebraic type: in a \texttt{send} operation or in a \texttt{receive} operation. For the former, we can use any expression of the correct basic type as an argument, because in C++ the type \texttt{T} is implicitly convertible to the type \texttt{const T &}. For the latter, it is important that each basic type expression given as an argument must be assignable (i.e. non-const), so that the received values may be placed in their variables. These constraints are reflected in the constructors, but not to the extent that we should like. If the second constructor is used for an object that will be \texttt{received}, we would like a type error, but this code would compile. When it executes, the received values will attempt to be placed in an undefined memory space and a segmentation fault would result. Ideally we would like this error to be caught at compile time. We explore this topic further in Chapter 10.

The \texttt{is\_alg\_type} function is necessary for two reasons. The first is to be able to mark those types which we should ask to serialise and deserialise themselves (using the \texttt{write} and \texttt{read} methods), rather than using the object's entire memory representation over the wire, which in this case would be inappropriate, since the memory representation contains pointers rather than actual data. The second is because of typing restrictions. Recall that when performing the \texttt{receive} operation, the argument must be a non-const reference because we alter the variable during a \texttt{receive}. This is not the case here, since the object itself is not modified, only those that it points to. We would also like to be able to construct the algebraic type in a temporary object for brevity in all cases. This requires that the argument be a const reference. So for \texttt{receive} operations on algebraic types, the argument must be a const reference.

4.7 Implementing the Ninja Communication Model in UNIX

According to the Ninja specification, participants must not share any memory. In fact, they may comprise individual compilation units (object files). However for simplicity, in our prototype all participants belong to the same compilation unit. In order for the participants not to share memory spaces, they must run in different processes. Thus every participant spawned using the \texttt{spawn} method will create a new process using the UNIX system call \texttt{fork}.

However, communication procedures within an instance of a participant must share memory between each other. Thus we must use threads for running communication procedures, listening for channel activations, etc. The POSIX standard for threads (\texttt{pthreads}) provides a framework we can use for this purpose.
Ninja’s invocation model maps very well onto persistent sockets. When a participant is spawned and has a communication procedure for a channel, the participant will begin to listen on the channel’s socket. When a channel is invoked, the invoking participant will connect to its socket, and use the newly opened communication link for session communication (by using the **read** and **write** system calls). By using the operating system for communication to as large an extent as possible, we avoid the need to implement our own communication system based on shared memory, semaphores etc. and use best-of-breed operating system features which have seen years of optimisation effort in real-world situations.

We must consider the different behaviours of the participant that are possible based on the lifetime of a channel it listens on. If the channel is linear, the participant may only listen on the channel once and will immediately start the communication procedure running in the same thread once the connection is made. If it is shared, the participant must spawn a new thread to run the communication procedure every time a connection is made, while continuing to listen in the original thread.

Some UNIX variants provide the ability to choose the CPU core on which a particular thread should run. This is known as CPU affinity (an **affinity mask** is a bitmask indicating the only cores on which the process shall run). On the Linux operating system, a thread may alter its affinity using the **sched_setaffinity** system call. By default (as is the case with any UNIX variant) the operating system will choose the CPU on behalf of the thread. This behaviour is that used by a participant when the default location, **os::default_location**, is given as a parameter to **spawn**. On Linux, if **os::cpu(x)** is given as a parameter to **spawn**, the participant shall set its affinity mask to only run on CPU x.

The UNIX socket mechanism is used to provide a communications area ‘in’ the filesystem (though of course in reality communication does not use the filesystem at all). When a channel is allocated, it picks a number x that has not been used before for a UNIX socket, and uses /tmp/socket−x for socket communication. When new processes (participants) are spawned, they will retain the same number x, and despite running in different memory spaces, the processes will be able to use the filesystem, and the UNIX sockets it contains, as a common namespace.

There are alternative UNIX implementations of the communication model we may have considered, these are discussed in Chapter 10.
Chapter 5

Delegates in Ninja

This chapter describes an extension to the Ninja language that introduces the concept of delegates to the language. Delegates were first described by [17] as “methods that can be used to satisfy part of a session”, but the author did not go into any more detail than this. We give a formulation for constructing generic delegates that can be used in any circumstance without violating type safety.

In this chapter, we shall define a delegate as a procedure that, given a session, manipulates it in some way (that is, by performing communication over it) and returns the resultant session. For such a delegate to be universally useful, it must support a wide range of session types as input. Clearly, there must be certain restrictions placed on the session types it accepts as input, namely that it must incorporate the communication actions performed by the delegate. However the resultant session type is defined by the user of the delegate. It incorporates those actions that are performed over the session after the delegate has been used. Since the output session type is derived from the input session type, we conclude that the input session type must contain the output session type as part of its definition. So the input session type may be obtained by performing a well-defined transformation on the output session type. Thus, the definition of a delegate comprises not only its communication procedure but also a function over session types which transforms the output session type into the input session type.

```c
typedef session addone {
    in int;
    out int;
    Cont;
} Addone(Cont);
```

Figure 5.1: A Ninja parameterised typedef for an add-one delegate
CHAPTER 5. DELEGATES IN NINJA

```cpp
template <class Cont>
Cont addone(Addone(Cont) s) {
    int i;
    receive(s, i);
    ++i;
    send(s, i);
    return s;
}
```

Figure 5.2: The Ninja communication procedure for an add-one delegate

```cpp
typedef session repaddone {
    Addone(
        in { ok(); end | again(); Repaddone
        }
    )
} Repaddone;
```

Figure 5.3: Definition of a Ninja session type that uses the add-one delegate

How may we add this feature to the Ninja language? Firstly we must introduce the idea of functional typedefs, essentially typedefs which may be parameterised using one or more parameters. The C++ language does not allow for functional typedefs per se, but we may simulate them using typedefs embedded in template classes. For our purposes, we shall assume typedefs may be parameterised as a first-class feature of the language. An example of a functional typedef that may be used in the definition of an add-one delegate is shown in Figure 5.1. The communication procedure of the add-one delegate may now be introduced as a template-parameterised function as shown in Figure 5.2.

This delegate is now ready to be used by clients. First of all, the client

```cpp
Repaddone s = ...;
while (1) {
    s = addone(s);
    receive(s) {
        ok(): break;
        again(): continue;
    }
}
```

Figure 5.4: Usage of the Ninja add-one delegate
must incorporate the communication actions of the delegate into its session type. This is easily done using the functional definition of the delegate’s session type. In the case of our add-one delegate, an example of such a session type is shown in Figure 5.3. The client may then call the delegate’s communication procedure at any point at which the session’s type is in the range of the delegate’s transformation function. After the communication procedure has completed, the client may assume that the inverse of the transformation function has been applied to the session’s type. An example of such a usage is shown in Figure 5.4.

5.1 Extensions

There are many ways in which we can extend the idea of parameterisation to build more generic delegates. The most trivial way is to add arguments to the communication procedure. This is possible as it is just an ordinary C++ function. A more powerful technique is to parameterise delegates over simple types. This allows us to build delegates that may be specialised for speed (e.g. by specifying a low bitwidth integer type for a numeral type parameter) or for accuracy (e.g. by specifying a high bitwidth fixed-point or floating-point type for the numeral type parameter). This would entail adding parameters to the transformation function; the parameters for the communication procedure would be implicit in its definition as the type of its parameter incorporates the transformation function arguments.

A yet more powerful technique would be to parameterise the delegate over multiple different continuations. This would be useful if the delegate needs to make some form of decision that would necessarily affect the continuation type that the client would use (for example, if one of a delegate’s tasks is deciding whether communication shall continue). The transformation function would need to have as many arguments as there are possible results of the decision. The decision it makes may be inferred from the type of the session that it eventually returns. The fact that the return type of the delegate is non-deterministic needs to be reflected in its usage by the client, and fundamentally by the language itself. A switch-like statement may be employed, similar to the complex form of the receive primitive, to choose between the different possibilities.

We may construct a delegate that operates on two or more sessions. In this case, the number of output sessions is equal to the number of input sessions, which is equal to the number of continuation types. Many languages do not easily support more than one return value, so for these cases the output session types are returned via call-by-reference parameters to the delegate’s communication procedure.

5.2 Implementing Delegates in Pure C++

We have defined a delegate as the tuple of a parameterised session typedef and a parameterised communication procedure. Both are parameterised over the
chapter 5. delegates in ninja

\[
src(s, c, d) = \begin{cases} 
  d & \text{if } s = \text{call}(c) \\
  \text{rc}(s) & \text{otherwise}
\end{cases}
\]

Figure 5.5: The \texttt{special\_remove\_call} function

same set of types, i.e. zero or more simple types and one or more continuations. We can adapt this to the C++ language by defining a delegate as a function which returns two values: the session type and the communication procedure. Due to time constraints and the lack of an ‘intuitive’ non-deterministic return type in the C++ language, a delegate may only sensibly be parameterised over one continuation, however we may have as many simple types as we wish. We may also parameterise over values, which could prove useful in some cases such as a delegate which can repeat its actions an arbitrary number of times (this is necessarily reflected in the session type).

We must make a special concession for delegates which may be placed in a loop consisting only of that delegate. As we have seen, for the \texttt{unroll} operation to work properly, the rolled-up type must be fully defined. The delegate must perform the \texttt{unroll} operation as part of deriving the final continuation type, if the continuation type given to the delegate is of the form \texttt{call} \texttt{<}c\texttt{>} (this is represented by the \texttt{rc}, or \texttt{remove\_call}, function given in Figure 4.1). However in C++, when a template is instantiated, the instantiation of the template is placed before the first statement in which it is used. Normally this will be between lines 1 and 2 of the definition of the session type using the protocol defined in Figure 3.1, at which stage the rolled-up type will not be fully defined. Usually we can work around this problem, simply by adjusting the session type so that the unroll step does not occur immediately after any delegate is used. However for a loop consisting only of a delegate we cannot work around this problem, since we have only one place to put the unroll step: directly after the delegate. Therefore we cannot use our usual techniques for eliminating unrolls.

Instead we define a \texttt{special\_remove\_call} function, an extension of \texttt{remove\_call}, which takes two additional parameters. The first is the name of a rolled-up type. The second is its unrolled definition. Users of a delegate will supply the name of a rolled-up type to the delegate if the unrolled version of the type will be defined in the future to be the delegate’s session type (a dummy type will be supplied in the normal case, where the delegate is not in a tight loop). Delegates, in order to find the final continuation type, will call the function giving the user-supplied rolled-up type as the first parameter, and the delegate’s session type as the second. A functional definition of \texttt{special\_remove\_call} is given in Figure 5.5.

An example of a pure C++ delegate, which has been translated from the Ninja delegate of Figures 5.1 and 5.2, is shown in Figure 5.6.
template <class Cont, class RType = void>
struct repaddone {

typedef seq<in<int>, seq<out<int>, Cont> > t;

typedef special_remove_call<Cont, RType, t>::t ct;

session<ct> addone(session<t> s) {
    int i;
    session<seq<out<int>, Cont> > = s.receive(i);
    session<ct> s2 = s1.send(i+1);
    return s2;
}
};

Figure 5.6: C++ definition of the Ninja add-one delegate of Figures 5.1 and 5.2
Chapter 6

Case Studies

The following are examples of relatively substantial protocols which have been implemented in our language. The discussion should hopefully convince you of the language’s flexibility.

6.1 A Publish/Subscribe Broker

A publish/subscribe broker is a software component that manages communications amongst other components. A component may send a subscription request to the broker, requesting it be sent messages of a certain type. From that point on, any messages of that type that are published by sending them to the broker are forwarded to the subscribers. Publish/subscribe brokers have seen a variety of implementations, such as the MQTT [2] broker.

We shall construct a participant which implements a simple broker that is responsible for three types of message. A participant may subscribe to any of these three message types. Participants will communicate with the broker over a single channel that is provided for this purpose.

In order for the broker to be able to deal with multiple clients in a single communication procedure, a dynamic channel will be used. Since we will both be expecting input and sending output on a session at the same time, the initial session type is of the forked type. The first forked session deals exclusively with input, and the second with output. In the initialisation procedure for the channel, the session is forked, and a mapping is built from input session handles to output session handles. The input session handle is used as the stored type.

Three subscriber lists are maintained, one for each message type. A subscriber list is a list of session handles of the output session type. The message types themselves are algebraic types of the names Msg1, Msg2 and Msg3, each taking a 1-tuple int as data. To subscribe to a message type, an algebraic type Subscribe is provided, with a 1-tuple int as data. The data in this case is the type identifier of the algebraic type to which you wish to subscribe. When a subscription request is received, the output session handle for the participant is
looked up in our mapping from input handles to output handles, and added to the appropriate subscriber list. When a message of type Msg1-3 is received, the message is forwarded on along the output session handles of all subscribers to that particular message type.

We have established that the input session must handle subscription requests as well as incoming broadcasts. The output session must handle outgoing broadcasts. So the complete session type for the publish/subscribe broker is

\[
\text{pubsub\_in = choice< seq< in< Subscribe >, pubsub\_in >,}
\text{ seq< in< Msg1 >, pubsub\_in >,}
\text{ seq< in< Msg2 >, pubsub\_in >,}
\text{ seq< in< Msg3 >, pubsub\_in > >}
\]

\[
\text{pubsub\_out = choice< seq< out< Msg1 >, pubsub\_out >,}
\text{ seq< out< Msg2 >, pubsub\_out >,}
\text{ seq< out< Msg3 >, pubsub\_out > >}
\]

\[
\text{pubsub = forked< pubsub\_in, pubsub\_out >}
\]

A full code listing for the broker follows.

```c
#include <sessiontypes>
#include <set>
#include <map>
#include <unistd.h>

using namespace std;

DECLARE_ALG_TYPE(32, Subscribe, 1, (int));
DECLARE_ALG_TYPE(33, Wait, 0, ());
DECLARE_ALG_TYPE(36, Msg1, 1, (int));
DECLARE_ALG_TYPE(37, Msg2, 1, (int));
DECLARE_ALG_TYPE(38, Msg3, 1, (int));

struct pubsub\_in\_r;
typedef choice< seq<
in< Subscribe >,
call< pubsub\_in\_r > >,
seq<
in< Msg1 >,
call< pubsub\_in\_r > >,
seq<
in< Msg2 >,
call< pubsub\_in\_r > >,
seq<
in< Msg3 >,
call< pubsub\_in\_r > > > pubsub\_in;

struct pubsub\_in\_r { typedef pubsub\_in t; };
```
6.1. A PUBLISH/SUBSCRIBE BROKER

```c
struct pubsub_out_r;
typedef choice< seq<
  out<Msg1>,
  call<pubsub_out_r>,>
seq<
  out<Msg2>,
  call<pubsub_out_r>,>
seq<
  out<Msg3>,
  call<pubsub_out_r>> > pubsub_out;
struct pubsub_out_r { typedef pubsub out t; };
typedef forked<pubsub_in, pubsub_out> pubsub;
struct broker_base {
  typedef set<session_handle<pubsub_out>> handleset;
  handleset msg1_subscribers,
    msg2_subscribers,
    msg3_subscribers;
  map<session_handle<pubsub_in>, session_handle<pubsub_out>> in_out_map;
  void init() {} 

  session_handle<pubsub_in> ch1new(session<pubsub> s) {
    session<pubsub_in> sin;
    session<pubsub_out> sout;
    s.fork(sin, sout);

    session_handle<pubsub_in> hsin = sin;
    in_out_map[hsin] = sout;
    return hsin;
  }

  void ch1(session_list<pubsub_in>& sl) {
    int index, msg_data;
    choose_session(pubsub_in, sh, sl) {
      session<pubsub_in> s = *sh;

      session_choice(session<pubsub_in>, s) {
        session_choice_case_nodecl (s, Subscribe(index)) {
          sh.release(s);
          switch (index) {
            case MSG_TYPEID(Msg1(0)):
              msg1_subscribers.insert(in_out_map[sh]);
            break;
            case MSG_TYPEID(Msg2(0)):
              msg2_subscribers.insert(in_out_map[sh]);
          }
        }
      }
    }
  }
```
break;
case MSG_TYPEID(Msg3(0)):
    msg3_subscribers.insert(in_out_map[sh]);
    break;
}
}

session_choice_case_nodecl(s, Msg1(msg_data)) {
    sh.release(s);
    for (handleset::iterator i = msg1_subscribers.begin();
            i != msg1_subscribers.end();
            ++i) {
        session<pubsub_out> s = *(i);
        s = s.send(Msg1(msg_data));
        i->release(s);
    }
}

session_choice_case_nodecl(s, Msg2(msg_data)) {
    sh.release(s);
    for (handleset::iterator i = msg2_subscribers.begin();
            i != msg2_subscribers.end();
            ++i) {
        session<pubsub_out> s = **i;
        s = s.send(Msg2(msg_data));
        i->release(s);
    }
}

session_choice_case_nodecl(s, Msg3(msg_data)) {
    sh.release(s);
    for (handleset::iterator i = msg3_subscribers.begin();
            i != msg3_subscribers.end();
            ++i) {
        session<pubsub_out> s = **i;
        s = s.send(Msg3(msg_data));
        i->release(s);
    }
}

session_choice_end;
}
typedef participant<broker_base, 
    dynamic_channel<pubsub, pubsub_in>, 
    &broker_base::ch1, 
    &broker_base::ch1new 
> broker;

struct publisher_base {
    channel<pubsub> *ch1;

    void init() {
        session<dual<pubsub>::t> s = ch1->invoke();

        session<dual<pubsub_in>::t> sin;
        session<dual<pubsub_out>::t> sout;
        s.fork(sin, sout);

        for (int i = 0; i < 10; i++) {
            usleep(90000);
            cout << "[Publisher] Iteration " << i << endl;
            sin = sin.send(Msg1(i));
            sin = sin.send(Msg2(i));
            sin = sin.send(Msg3(i));
        }
    }
};

typedef participant<publisher_base, 
    dual_channel<pubsub>, &publisher_base::ch1 
> publisher;

struct subscriber_base {
    channel<pubsub> *ch1;

    int id, subscription;

    void init() {
        session<dual<pubsub>::t> s = ch1->invoke();

        session<dual<pubsub_in>::t> sin;
        session<dual<pubsub_out>::t> sout;
        s.fork(sin, sout);
sin = sin.send(Subscribe(subscription));
cout << "[Subscriber−" << id << "] Sent subscription"
<< subscription << endl;
while (1) {
    int data;
    session_choice(session<dual<pubsub_out>::t>, sout) {
        session_choice_case_nodecl(sout, Msg1(data)) {
            cout << "[Subscriber−" << id << "] Received Msg1(" << data << ")" << endl;
        }
        session_choice_case_nodecl(sout, Msg2(data)) {
            cout << "[Subscriber−" << id << "] Received Msg2(" << data << ")" << endl;
        }
        session_choice_case_nodecl(sout, Msg3(data)) {
            cout << "[Subscriber−" << id << "] Received Msg3(" << data << ")" << endl;
        }
    }
    session_choice_end;
}
}

typedef participant<subscriber_base,
dual_channel<pubsub>, &subscriber_base::ch1
> subscriber;

int main() {
    channel<pubsub> ch;
    broker br(ch);
    publisher pb(ch);
    subscriber s1(ch),
    s2(ch),
    s3(ch),
    s4(ch);
    s1.id = 1;
    s1.subscription = MSG_TYPEID(Msg1(0));
6.2. AN ELECTION PROTOCOL

[14] gives a protocol for a connected ring of \( n \) processes, all of which are identical, to elect a leader amongst themselves, given that each process has an independent random number generator. A slightly modified version of the protocol is shown in Algorithm 1. We shall describe how to implement this protocol as a delegate in our language.

In order to give this delegate a session type, we must first analyse its behaviour with regards to communication. We can see that the program will send \( n - 1 \) messages of type \texttt{bool}, and then make a decision about whether to continue or stop (note that all participants will make the same decision). For the first part of the session type, sending \( n - 1 \) \texttt{bool} messages, we can use the \texttt{repeat} function in order to generate a session type that repeats sending the \texttt{bool} the given number of times. For the second part of the type, the decision, we may na"ively assume that we do not need to send any more messages, since all participants will come so the same conclusion. However this does not fit our typing mechanism. In order to make a decision, a participant must send a value...
Algorithm 1 The ring leader election protocol

\[
\begin{align*}
&n \leftarrow \text{number of processes in ring} \\
&\text{eligible} \leftarrow \text{true} \\
&\text{count} \leftarrow n \\
\text{while } n \neq 1 \text{ do} \\
&\text{oldcount} \leftarrow \text{count} \\
&\text{if } \text{eligible} \text{ then} \\
&\quad r \leftarrow \text{rand}(0 \ldots \text{count} - 1) \\
&\quad \text{if } r = 0 \text{ then} \\
&\quad \quad \text{candidate} \leftarrow \text{true} \\
&\quad \text{else} \\
&\quad \quad \text{candidate} \leftarrow \text{false} \\
&\quad \text{end if} \\
&\quad \end \text{if} \\
&\quad \text{else} \\
&\quad \quad \text{candidate} \leftarrow \text{false} \\
&\quad \text{end if} \\
&\quad \text{othercandidate} \leftarrow \text{candidate} \\
&\quad \text{count} \leftarrow 0 \\
\text{for } i = 1 \text{ to } n - 1 \text{ do} \\
&\quad \text{if } \text{othercandidate} \text{ then} \\
&\quad \quad \text{count} \leftarrow \text{count} + 1 \\
&\quad \text{end if} \\
&\quad \text{send } \text{othercandidate} \text{ clockwise} \\
&\quad \text{receive } \text{othercandidate} \text{ anticlockwise} \\
\text{end for} \\
&\text{if } \text{othercandidate} \text{ then} \\
&\quad \text{count} \leftarrow \text{count} + 1 \\
&\text{end if} \\
&\text{if } \text{count} = 0 \text{ then} \\
&\quad \text{count} \leftarrow \text{oldcount} \\
&\text{else} \\
&\quad \text{eligible} \leftarrow \text{candidate} \\
&\text{end if} \\
\text{end while} \\
\text{return } \text{candidate}
\end{align*}
\]
of a particular type that matches one of the available choices. Suppose the first action supplied by our delegate's continuation type was to send a bool. By sending a bool, the typing mechanism cannot establish which of the two paths should be taken. So when the decision is made, we first send a dummy value, a 0-tuple contained in an algebraic type with one of two names: ElectionContinue or ElectionStop, with obvious semantics. The ElectionContinue action will recursively invoke the delegate's session type, and ElectionStop will proceed to the continuation type.

The full session type of the delegate with \( N \) processes in the ring and a continuation type of \( \text{Cont} \) is:

\[
t = \text{repeat}<N-1, \text{out}<\text{bool}>, \\
\quad \text{choice}<\text{seq}<\text{out}<\text{ElectionContinue}>, t>, \\
\quad \text{seq}<\text{out}<\text{ElectionStop}>, \text{Cont} > > :: t
\]

### 6.2.1 Implementation

Implementation is complicated by the fact that each iteration of the loop will use sessions of different types. Thus each iteration is implemented separately. The \( \text{count_candidates} \) function is parameterised over a count of remaining iterations \( \text{CurCount} \), the total number of iterations and the continuation type. It provides a method \( f \) that performs the last \( \text{CurCount} \) iterations of the loop and returns the cumulative candidate count \( \text{count} \). The method \( f \) operates by performing one (correctly typed) iteration of the loop and calling the instance of \( f \) specified with the same parameters but for a remaining iteration count of \( \text{CurCount} - 1 \). The function \( \text{count_candidates} \) for a \( \text{CurCount} \) value of 1 is specialised such that its \( f \) method will only perform one iteration and return immediately.

The definition of the function that returns the session types for the delegate is placed before the definition of both the delegate and \( \text{count_candidates} \). The function that produces the session types is separated from the delegate because \( \text{count_candidates} \) needs to be able to use these definitions.

The delegate \( \text{election} \) takes two sessions as input: the output session, which is used to send data clockwise to the next participant, and the input session which is used to receive data anticlockwise from the previous participant. It returns two sessions: the resultant output and input sessions, and a bool indicating whether this delegate was elected as a leader.

We cannot create a perfect ring easily in our language. Consider five identical participants linked together as in Figure 6.1. In order for communication to begin, one participant must be invoked. This participant can then invoke the others. Which participant should invoke the next one? We don’t know, since they are all identical. Another solution may be to invoke the next participant in the initialisation phase. However this would lead to a deadlock situation, since all initialisation is performed before any participant begins to listen on its channels.
Figure 6.1: A five-participant ring

Figure 6.2: A five-participant ring with a kicker
6.2. AN ELECTION PROTOCOL

Our solution is to create a “kicker” participant, which is linked to an arbitrary ring member via a dummy channel (no communication happens over this channel). The channel layout is shown in Figure 6.2. The kicker will invoke its channel upon initialisation, which will cause the dummy channel’s communication procedure to run. The procedure will invoke the next participant and store the session in a field of the participant. Each participant will invoke the next one when it is invoked. This will happen until we come full circle and the participant we kicked is invoked. Instead of invoking the next participant, the communication procedure will use the session that was stored by the dummy communication procedure.

A full code listing for the election protocol follows.

```cpp
#include <sessiontypes>
#include <iostream>
#if defined unix
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#endif

using namespace std;

DECLARE_ALG_TYPE(32, ElectionContinue, 0, ());
DECLARE_ALG_TYPE(33, ElectionStop, 0, ());

template <typename Cont, class RType = void>
struct election_types {

    struct t_r:
        typedef choice<seq<out<ElectionContinue>, call<t_r>>,
                       seq<out<ElectionStop>, Cont>> econt;
    typedef typename repeat<Cont-1, out<bool>, econt>::t t;
    typedef t t_r;
    struct t_r { typedef t_t t; }
};

template <typename Cont, class RType = void>
struct count_candidates {

typedef election_types<Cont, RType> et;

static int f(
    session<typename repeat<CurCount-1, out<bool>>,
                typename et::econt>::t &sout1,
    session<typename dual<typename repeat<CurCount, out<bool>>,
```
CHAPTER 6. CASE STUDIES

```cpp
typedef typename et::econt >::t &sin,
  session<typename et::econt> &soutn,
  session<typename> dual<typename et::econt> >::t &sinn
} {
  bool b;

  session<typename> dual<typename repeat<CurCount−1, out<bool>>,
  typename et::econt >::t ::t sin1 = sin.receive(b);
  session<typename repeat<CurCount−2, out<bool>>,
  typename et::econt >::t sout2 = sout1.send(b);

  return (b ? 1 : 0) + count_candidates<CurCount−1, Count,
  Cont, RType>::f(sout2, sin1, soutn, sinn);
}

};

template <int Count, class Cont, class RType>
struct count_candidates <1, Count, Cont, RType> {

  typedef election_types <Count, Cont, RType> et;

  static int f(
    session<typename et::econt> &sout1,
    session<typename> dual<seq<out<bool>>,
    typename et::econt > ::t &sin,
    session<typename> et::econt> &soutn,
    session<typename> dual<typename et::econt> >::t &sinn
  ) {
    bool b;

    sinn = sin.receive(b);
    soutn = sout1;

    return b ? 1 : 0;
  }
}

);}

#endif unix

void seed_random () {
  unsigned int seed;

  int fd = open("/dev/urandom", O_RDONLY);
  read(fd, &seed, sizeof(seed));
  close(fd);
```
6.2. AN ELECTION PROTOCOL

```cpp
srnd(seed);
}
#endif

template <typename et, class Cont, class RType = void>
struct election {

typedef election_types<Count, Cont, RType> et;
typedef typename special_remove_call<Cont, RType, typename et::t>::t rcont;

static bool f(
    session<type> sout,
    session<type> dual<type> et::t sin,
    session<rcont> &sout_out,
    session<type> dual<rcont>::t &sin_out
) {
    srand_random();
    int cand_count = Count;

    bool is_candidat = true;
    while (1) {
        is_candidat = is_candidat &&
            (rand() < RANDMAX/cand_count);
        cout << "is_candidat = " << is_candidat << endl;
        typename c_session<type> et::t,
        out<bool> et::t sout1 = sout.send(is_candidat);

        session<type> et::tcont soutn;
        session<type> dual<type> et::tcont sin;
        cand_count = is_candidat +
            count_candidates<Count-1, Count, Cont, RType>::f(sout1, sin, soutn, sin);

        if (cand_count == 1) {
            if (is_candidat) {
                cout << "I won the election" << endl;
            } else {
                cout << "I lost the election" << endl;
            }
            sout_out = soutn.send(ElectionStop());
            sin_out = sin.receive(ElectionStop());
            return is_candidat;
        } else {
            cout << "We could not agree on a leader" << endl;
            sout = soutn.send(ElectionContinue());
        }
    }
}
```
\begin{verbatim}

sin = sinn.receive(ElectionContinue());
if (cand_count == 0) {
    cand_count = Count;
    is_candidate = true;
}
}

struct ring_dummy_r;
typedef seq<in<int>, call<ring_dummy_r>> ring_dummy;
struct ring_dummy_r { typedef ring_dummy t; }

template <int Count>
struct ring_cpu {

typedef election<Count, ring_dummy> el;
typedef typename dual<typename el::et::t>::t t;

struct base_type {

    base_type() : use_csofar(false) {} 

    channel<t> *ch1;

    bool use_csofar;

    session<typename el::et::t> csofar;

    void ch2(session<t> sin) {
        session<typename el::et::t> sout;
        if (use_csofar) {
            sout = csofar;
        } else {
            sout = ch1->invoke();
        }
    }

    session<ring_dummy> sout_out;
    session<typename dual<ring_dummy>::t> sin_out;
    bool is_master = el::f(sout, sin, sout_out, sin_out);
    cout << "is_master = " << is_master << endl;
}
\end{verbatim}
6.2. AN ELECTION PROTOCOL

```c
void ch3(session<ring_dummy> skick) {
    use_cso2ut = true;
    cso2ut = ch1->invoke();
}

void init() {}
};

typedef participant<base_type,
    dual_channel<t>, &base_type::ch1, 0,
    linear_channel<t>, &base_type::ch2, 0,
    linear_channel<ring_dummy>, &base_type::ch3,
> participant;
}

struct kicker_base {

    channel<ring_dummy> *ch1;

    void init() {
        ch1->invoke();
    }
};

typedef participant<kicker_base,
    dual_channel<ring_dummy>, &kicker_base::ch1,
> kicker;

typedef ring_cpu<5> pentagon;

int main() {
    channel<pentagon::t> ch1, ch2, ch3, ch4, ch5;
    channel<ring_dummy> d1, d2, d3, d4, d5;
    pentagon::participant
        p1(ch1, ch2, d1),
        p2(ch2, ch3, d2),
        p3(ch3, ch4, d3),
        p4(ch4, ch5, d4),
        p5(ch5, ch1, d5);

    kicker k1(d1);

    p1.spawn(os::default_location);
```
```cpp
p2.spawn(os::default_location);
p3.spawn(os::default_location);
p4.spawn(os::default_location);
p5.spawn(os::default_location);
k1.spawn(os::default_location);

os::sleep(1);
```
Chapter 7

Statically Verifying the Linearity Constraint

This chapter describes how the linearity constraint is verified using ROSE’s program analysis framework.

7.1 Classical Program Analyses

The classical program analysis formulation is given in [18]. It defines a monotone framework which can be used to solve a number of program analysis problems. The essential equations of the monotone framework are given in Figure 7.1.

A classical analysis may either be a forward or a backward analysis. Both analyses use the set of arcs in the control flow graph of the program $p$, known as $\text{flow}(p)$. Forward analyses use $\text{flow}(p)$ as $F$, and the initial (guaranteed to be executed first) statement of the program, $i$, is the only element of the set of extremal statements $E$. Backward analyses use the reverse flow of $p$, $\{(l_2, l_1) | (l_1, l_2) \in \text{flow}(p)\}$ or $\text{flow}^R(p)$ as $F$ and the final statements (only one of these statements is guaranteed to be executed last) as the extremal statements $E$.

Analyses define the type $T$ of the data associated with each statement, the transfer function $f_x$ of type $T \rightarrow T$ for each statement $x$, the join function $\sqcup$ of type $(T \times T) \rightarrow T$, the extremal data $\iota$ which is used to bootstrap the algorithm at the extremal statements and the initial data $\bot$ used to bootstrap the algorithm at all other statements.

\[
\begin{align*}
\text{Analysis}_{\cup}(x_2) &= \left\{ \bigcup_{i} \{\text{Analysis}_{\cup}(x_1) | (x_1, x_2) \in F\} \right\} \text{ if } x_2 \notin E \\
\text{Analysis}_{\cdot}(x) &= f_x(\text{Analysis}_{\cdot}(x))
\end{align*}
\]

Figure 7.1: Monotone framework equations
int x = 1, y;
while (1) {
  y = x;
}

Figure 7.2: A program whose variable $x$ has an infinite use count.

In order to solve these equations for every statement $x$, we use an iterative approach. Initially the solution to all equations for non-extremal statements is set to $\bot$, and the solution for extremal statements is set to $\iota$. The new solutions are then computed using the dataset we have just established. This process is repeated until no solution changes.

In order to ensure that the above algorithm will terminate, for every statement $x$, for every chain of solutions $s_1, s_2, \ldots, s_n$ such that $s_{i+1} = f_x(s_i)$, there must be a $k (1 \leq k < n)$ such that $s_k = s_{k+1}$. This is known as the ascending chain condition.

### 7.2 The UseCount Analysis

The UseCount analysis is an instance of the monotone framework. It aims to find, for every program point, the maximum number of times that every variable, from that point on, will be used until it is redefined. The UseCount analysis is very useful to us for determining linearity. We simply need to verify that every linear variable at every program point shall be used no more than once until it is redefined.

It is important to impose a maximum value on every variable’s use count. Consider the program shown in Figure 7.2. Since the loop shown between lines 2 and 4 is infinite, we may simplify the monotone framework equations to $x = f_2(f_3(x))$. Since $f_2$ has no effect (line 2 does not use or redefine any variables), we may simplify this equation to $x = f_3(x)$. However, this equation has no solution if we do not impose a maximum, since $f_3$ will always increment $x$’s use count by 1. Therefore the unconstrained version of the analysis does not satisfy the ascending chain condition and will never terminate. If we do impose a maximum $m$, $f_3^m(x) = f_3^{m+1}(x)$, the analysis satisfies the ascending chain condition and thus will terminate.

UseCount is a backward analysis. Its data type is $\text{Var} \rightarrow \mathbb{N}$ and its transfer functions, join function, extremal data and initial data are defined functionally in Figure 7.3.

For the linearity analysis, we only care about three possible use counts: if the use count for a variable at a particular point is 0 or 1, the variable satisfies the constraint at that point; if the use count for a linear variable is 2 or above, we have failed the linearity check. If the use count is greater than 2, we are led to the same conclusion as if it were 2, so we can safely use a maximum use count of 2 for linearity checking purposes.
7.3 Implementing UseCount in ROSE

ROSE provides a data flow analysis module that generically implements the monotone framework analysis. It also provides a sample analysis – the reaching definition analysis. In order to construct an analysis, the DataFlowAnalysis template is instantiated with the following parameters:

- the type of the control flow node. This class implements the transfer function, and stores the entry and exit data. Most of this is done for you by a parameterised class DataFlowNode, from which the node class must inherit;
- the analysis’s data type (in this case, we use an STL map from string to int);
- the direction of the analysis (forward or backward)\(^1\).

The parameterised DataFlowAnalysis is then extended to produce an analysis class. The analysis class implements the meet (sic, should be join) function. In our case, it also stores the maximum use count.

Why do we use the string class to represent variables, rather than some AST variable declaration class? This is in conformance with the other data flow analyses which use variable names to represent variables, appending a unique scope identifier to distinguish between variables in different scopes, and which may provide extended variable information elsewhere. For our analysis a variable name to type mapping is provided in the analysis class, in order to distinguish between linear and nonlinear variables.

ROSE’s data flow analysis nodes do not represent individual statements, but rather basic blocks (a basic block is a block of statements which will always be

\[^1\text{Note that originally, this parameter did not exist. So the first order of business was to add the parameter.}\]

\[
f_x(d)(v) = \begin{cases} 
\min(\text{gen}(x)(v), m), & \text{if } v \in \text{kill}(x) \\
\min(d(v) + \text{gen}(x)(v), m), & \text{otherwise} 
\end{cases}
\]

\[
\text{gen}(x)(v) = \text{the number of times } v \text{ is used in statement } x
\]

\[
\text{kill}(x) = \text{the set of variables that are defined in } x
\]

\[
(d_1 \sqcup d_2)(v) = \max(d_1(v), d_2(v))
\]

\[
\min(x, y) = \begin{cases} 
x, & \text{if } x < y \\
y, & \text{otherwise}
\end{cases}
\]

\[
\max(x, y) = \begin{cases} 
x, & \text{if } x > y \\
y, & \text{otherwise}
\end{cases}
\]

\[
\iota(v) = 0
\]

\[
\bot(v) = 0
\]
7.4 Using UseCount to Implement a Linearity Checker

After implementing UseCount, implementing an analysis that uses it is relatively trivial. Since ROSE acts like a compiler, after running the frontend we have
available to us the ASTs of the source files that are given on the command line. We can write an AST traversal which searches for function definitions within the ASTs, and invokes the UseCount analysis on these function definitions with a maximum use count of 2. For each basic block $x$ and variable $v$, we examine the incoming, internal and outgoing use counts of $v$ under $x$. If a linear variable is found whose use count is equal to 2, the check fails and we give a diagnostic message to the user.

To determine if a variable is linear, we look up its type using the variable name to type mapping available to us in the analysis class. We then pass the AST representation of the type to a function `isLinear`, which returns a `bool` result. Currently, the `isLinear` function only checks whether the type is an instance of `session`, but this function could be extended to provide a much more powerful linear type system; see the next section.

### 7.5 Extensions

One of the most powerful extensions we can implement is a generic linear type system for C++. The concept of a linear type system was explored for functional languages in [28], however this is the first implementation known to this author for an imperative language. The best way of implementing a generic linear type system is to introduce annotations which we may use to indicate that a certain type is linear. There are two ways we can go about this:

1. a `type` modifier `linear`, similar to `const` and `volatile`;
2. a `class` modifier, which is part of the class definition itself.

The advantage of the first technique is that we can apply linearity to any type, without needing to make modifications to the type definition itself. However we need to consider whether this is a sensible idea. If a type is linear, that fact is reflected in its behaviour, and thus is a fundamental part of its design. Simply applying linearity to an existing type such as an `int` or STL string would give the object no useful semantics. In consideration of these facts, the best annotation strategy for linearity is a class modifier.

One important point to note about linear classes is that their inheritance hierarchies must be kept separate from non-linear inheritance hierarchies. We show that both allowing a nonlinear type to inherit from a linear type and vice versa would allow for the linear type system to be violated.

Firstly consider a nonlinear type $B$ that inherits from a linear type $A$. Since $B$ is nonlinear we may use an object of type $B$ in a nonlinear way, but this would violate the linearity constraint of its base type $A$.

Secondly consider a linear type $B$ that inherits from a nonlinear type $A$. An object of type $B$ must be treated linearly, but we can create a pointer to it. This pointer may easily be upcast to a pointer to an object of type $A$. The object referenced by this pointer may now be used in a nonlinear way, violating the linearity constraint.
Chapter 8

The Type Inference Mechanism

We describe a type inference system which allows us to infer the types of sessions, channels and participants in an untyped program. We also describe an implementation of this system, which has been implemented as a ROSE program transformation.

8.1 Theory

Our formulation of session types in this section shall be slightly different. Instead of the choice session type, we shall use sets. A session type is thus a set of seq elements. For representing the session type of a session variable $s$ we shall use $\sigma_s$.

A communication operation such as session $s1 = s.send(42)$; contains four useful pieces of information: the original session ($s$), the direction of information flow (out), the type of the datum (int) and the resultant session ($s_1$). Using this information we can derive a constraint on the original session type:

$$\text{seq(out(int), } \sigma_{s_1} \text{)} \in \sigma_s.$$

By taking all the communication operations present in a certain piece of code, we construct a system of simultaneous equations. For example, consider the communication procedure shown in Figure 8.1. From lines 4, 6 and 8 we derive the equations shown in Figure 8.2. After finding the equations we now wish to derive their minimal solution. In this case the solutions are trivially computed and are shown in Figure 8.3.

Why do we wish to find the minimal solution? One of the main motivating reasons is that identical recursive types which use differently named rolled-up types are treated as distinct types by the compiler. If we can equate identical types to as large an extent as possible, we ensure greater compatibility between communication procedures with the same session type in the same program.
void ch1(session s) {
  while (1) {
    int i;
    session s1 = s.receive(i);
    if (i+1 > MAX_INT) {
      s = s1.send(long(i+1));
    } else {
      s = s1.send(i+1);
    }
  }
}

Figure 8.1: Untyped communication procedure

seq(in(int), σs₁) ∈ σs
seq(out(long), σs) ∈ σs₁
seq(out(int), σs) ∈ σs₁

Figure 8.2: The set of constraints derived from Figure 8.1

σs = {seq(in(int), σs₁)}
σs₁ = {seq(out(long), σs), seq(out(int), σs)}

Figure 8.3: The solution to the set of constraints in Figure 8.2
8.1. THEORY

A crucial part of our analysis is the session graph. This is a directed graph whose nodes are session types, and whose edges are the actions that relate session types (an action edge is directed from the pre-action session type to the post-action session type). For example, the session graph for Figure 8.1 is shown in Figure 8.4.

The process of computing a minimal solution is that of unifying nodes. To unify a set of nodes we eliminate all but one of them, re-point all in-arcs and out-arcs to the one remaining node and eliminate all duplicate arcs. The most important part of node unification is deciding which nodes to unify. In general we must ensure the trace soundness and trace completeness of the resultant graph. A trace is a list of arc names, and a valid trace for a graph at node \( n \) is an empty trace, or a trace such that the first element of the trace has a corresponding out-arc from \( n \) to \( m \), and the remainder of the trace is a valid trace at \( m \). If a minimised graph is trace sound, any valid trace at any node \( n \) in the minimised graph is also a valid trace at any of the nodes in the original graph that were unified to create \( n \). If a minimised graph is trace complete, any valid trace at any node \( n \) in the original graph must also be a valid trace from the node that represents \( n \) in the minimised graph.

In the process of choosing nodes to unify, there are four main cases we need to concern ourselves with:

1. Identical divergent paths
2. Identical convergent paths
3. Identical disjoint subgraphs
4. Repetitive loops

Cases 1 and 2 are best illustrated by Figure 8.5. Nodes \( \sigma_{s_1} \) and \( \sigma_{s_3} \) are unifiable because they both diverge from \( \sigma_s \) via an arc \( \text{in}(t_1) \). Nodes \( \sigma_{s_2} \) and \( \sigma_{s_4} \) are also unifiable because they both converge to \( \sigma_{s_5} \) via \( \text{out}(t_4) \). The resulting session graph is shown in Figure 8.6.

Case 3 is the most commonly encountered case for communication procedures in the same program with compatible session types. Consider the graph shown in Figure 8.7. Nodes \( \sigma_s \) and \( \sigma_{s_2} \) are unifiable, and so are \( \sigma_{s_1} \) and \( \sigma_{s_3} \). The result is a single graph with the shape of the left-hand subgraph.
CHAPTER 8. THE TYPE INFERENCE MECHANISM

Figure 8.5: Session graph with identical divergent and convergent paths

Figure 8.6: Minimised version of Figure 8.5

Figure 8.7: Session graph with identical disjoint subgraphs
8.1. THEORY

```c
void ch1(session s) {
    while (1) {
        int i;
        session s1 = s.receive(i);
        session s2 = s1.send(i+1);
        session s3 = s2.receive(i);
        session s4 = s3.send(i+1);
        session s5 = s4.receive(i);
        s = s5.send(i+1);
    }
}
```

Figure 8.8: Example of a repetitive loop

![Figure 8.9: Session graph with repetitive loops](image)
bool can_unify(set(N) nodes) {
    return can_unify(nodes, nodes, true);
}

bool can_unify(set(N) first, set(N) nodes, bool isFirst) {
    if (size(nodes) <= 1) {
        return true;
    }
    if (!isFirst && first == nodes) {
        return true;
    }
    let out be the set of out-edges for every node in nodes
    — if some node in nodes does not have same set
    of out-edges as others, return false;
    for each edge o in out {
        newnodes = {};
        for each node n in nodes {
            follow edge o from n to m;
            add m to newnodes;
        }
        if (can_unify(first, newnodes, false) == false)
            return false;
    }
    return true;
}

Figure 8.10: Pseudocode implementation of unification decision algorithm

Case 4 is encountered where a program will perform the same actions multiple times in a loop (perhaps due to loop unrolling). Such a program is shown in Figure 8.8 and its session graph shown in Figure 8.9. We may unify nodes \( \sigma_{s_1}, \sigma_{s_2}, \sigma_{s_4} \) as well as nodes \( \sigma_{s_1}, \sigma_{s_3}, \sigma_{s_5} \).

Can we produce a unified algorithm to deal with all of these cases? Yes we can, apart from case 1. Our algorithm takes a set of nodes, and decides whether they are unifiable. Therefore we use the algorithm by constructing subsets of our graph, testing them against our algorithm and unifying if the algorithm succeeds. An outline of our algorithm is given in pseudocode in Figure 8.10.

Case 1 can be handled trivially. If a node has two or more identical out-edges, the nodes to which they point can be unified.

After we have produced a minimised graph, we can produce a node’s session type relatively easily using the node’s out-edges. If a node has 0 out-edges, its session type is \texttt{nil}. If a node has one out-edge \( o \) leading to a node \( n \) with session type \( t(n) \), its session type is \texttt{seq}<\texttt{\textless}o,t(n)\texttt{\textgreater}\. If a node has \( k > 1 \) out-edges \( o_1...k \)
8.1. THEORY

leading to nodes $n_1 \ldots n_k$, its session type is $\text{choice}<\text{seq}<\text{call}<\text{t}(n_1)>, \ldots, \text{seq}<\text{call}<\text{t}(n_k)>,>$. 

The problem with this algorithm is that it does not handle cycles in the graph. To solve this problem, each cycle has at least one node marked rolled-up. This node has a rolled-up name assigned to it. If this node is encountered while a session type is being generated, the node will report the session type $\text{call}<r>$, where $r$ is the rolled-up name of the session type. If the node’s type is requested directly, the rolled-up attribute is ignored and the session type is generated in the normal way (since an immediate session type cannot be of the form $\text{call}<r>$).

The unification process may have unintended consequences if we are not careful. Consider the program shown in Figure 8.11. Our algorithm would normally unify the types of $s$ and $s_3$ (call the result $\sigma_s$), as well as the types of $s_1$ and $s_2$ (call the result $\sigma_{s_1}$). However the type $\sigma_s$ is now incompatible with the dual of $\sigma_{s_1}$. The reason is that from one participant’s point of view, the rolled-up type is unrolled after 1 communication step and for the other, it is unrolled after 2 communication steps (which one is which depends on which node was marked ‘rolled-up’). So from the compiler’s point of view, these two types are incompatible (if the compiler were aware of our isorecursive type system, it could possibly equate the two types).

The solution to this problem is simple. Any session variable which is got from a use of the $\text{invoke}$ primitive is marked dual. Any session variable that is got from a dual variable is also marked dual. The representation of edges between dual nodes is flipped, so that $\text{in}<t>$ becomes $\text{out}<t>$ and vice versa (essentially performing the dual operation). When a dual variable’s type is required, we use the dual of that variable’s type rather than the type itself.

What consequences will this have for the program shown in Figure 8.11? Nodes $s$ and $s_2$ will be unified producing $\sigma_s$, and nodes $s_1$ and $s_3$ will be unified
producing $\sigma_s$. Now the type of $s$ will be compatible with the dual of the type of $s_2$.

8.2 Implementing the Type Inference Algorithm

The first step in implementing the algorithm is to create an untyped version of our C++ implementation of the Ninja language. Our untyped version of the language need not actually work, but it must compile. The reason for this is that ROSE expects any programs that are supplied to its frontend to be valid C++ programs. Creating an untyped version of the language entails creating versions of the `session` and `channel` templates and of the `linear_channel`, `shared_channel` and `dual_channel` functions 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 are untyped (known as the `untyped participants` variant).

The most efficient way of defining these variants is to use a macro-based system to switch between them. If a program uses the untyped sessions variant of the language, it should define the `UNTYPED_SESSIONS` macro before it includes the main library header file. If the program uses the untyped participants variant, it should define `UNTYPED_PARTICIPANTS` before including the header file. The presence of these macros will cause the library header file to define the untyped variants of the various classes and functions.

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 report they have not modified the graph. In the first stage, we unify identical divergent paths. In the second stage we perform all other unification using the algorithm shown in Figure 8.10. In order to make the code as concise as possible, the algorithm was
implemented in Prolog. This required that we find a suitable Prolog implement-
tion which provided a suitable language interface between C++ (or C) and
Prolog. In the end, SWI Prolog was chosen as our implementation.

The Prolog implementation of the algorithm must have access to the list
of arcs in the session graph. To this end, an \texttt{arc} predicate was exposed to
the Prolog code using SWI's Foreign Language Interface, such that
\texttt{arc(From, Arc, To)} holds if there exists an arc with direction and type information \texttt{Arc}
represented as a string) from the node \texttt{From} (represented as an internal pointer)
to the node \texttt{To} (also represented as a pointer). The Prolog implementation
produces a list of sets of nodes which must be unified. Since these nodes are
actually native pointers, we can convert them back to node pointers in order to
pass them to the node unifier.

After unification finishes, we must now mark nodes as rolled-up. This is
done in a way such that we aim to mark nodes which ‘start’ a session (i.e. the
return type of \texttt{invoke}, or the parameter type of a communication procedure)
as rolled-up, as is the usual practice.

Thirdly, we proceed to annotate the types of the session and channel (and
participant, if we are working with the untyped participant variant of the lan-
guage) variables with their types. Before we do this, however, all rolled-up
types must have their unrolled definitions placed at the top of the source file.
This is done by creating AST class and typedef declaration nodes and inserting
them into the global scope near the top of the file (we need to make sure the
definitions appear after we include the library header file though, since the un-
rolled definitions depend on them). Now we are ready to change the types of
the variables.
Chapter 9

Evaluation

9.1 Functionality Evaluation

The original goal of the project was to provide a framework that allows for provably correct programs to be written for multicore processors. This is functionally achieved via our session typing system. Most of the type safety proof is done by the compiler, which is an added bonus as no extra compilation steps are necessarily introduced to the user by our framework, unless we also wish to verify the linearity constraint.

Despite the immutable type system present in the C++ language, the usage of the framework would be most convenient for the user if a mutable type system were provided. The technique we have chosen allows almost complete implementation of the type system within the compiler (with the exception of the linearity constraint), but necessitates that unwieldy type specifications be used throughout the program unless external tools are used. These tools fulfill their purpose, but still require the user to declare a new session after every communication action. This adds overhead for the user, since the concept of having intermediate sessions have no meaning unless they are typed. It ought to be possible to write a transformation that allows us to transform a program written assuming mutable session types into one that uses our protocol of declaring a new session after every communication action (see chapter 10 for a discussion of how this might be done). From there, we can apply the type inference algorithm to derive a standard program in our language.

In terms of hardware implementation, we support multicore processors such as the Xeon which expose their features using standard operating system SMP mechanisms. Although we have implemented a working prototype for generic UNIX-like systems, we have no implementation for any specialised multicore processor such as Cell. It is important to have such an implementation because such specialised processors are becoming increasingly common, and a proof of concept would be useful to extend the concept to other such processors.
9.2 Performance Evaluation

We use two programs to profile the performance of our language. The first is a simple IO-bound request-reply example which simply bounces values between two participants. The second is our publish/subscribe broker. Both have been annotated with profiling code that counts the number of iterations per second of their main loops.

We shall run our experiments on a variety of systems, in a variety of modes. In order to negate the effects of other programs on our experiments, and as a matter of good experimental procedure, experiments shall be conducted in the following manner:

- wait until the system load average drops to below 0.1 before running any experiments;
- run the experiment multiple times, and average results.

The three systems we shall use for these tests are:

- System 1: Uniprocessor Intel(R) Pentium(R) 4 CPU 2.66GHz, 512KB cache, 256MB RAM, Linux 2.6.17
- System 2: 4-core Intel(R) Xeon(TM) CPU 2.00GHz, 512KB cache, 2GB RAM, Linux 2.6.13
- System 3: 2-core Intel(R) Pentium(R) D CPU 2.80GHz, 1024KB cache, 1GB RAM, Linux 2.6.18

We shall run tests in two different modes. In single-core mode all participants are constrained to run on the same core. In multi-core mode participants are constrained to different cores. The goal here is to observe the effects of communication between separate cores on system performance.

After performing some preliminary experiments I found that for the publish/subscribe broker, the iteration rate was extremely variant across multiple runs of the benchmark – the lowest iteration rate was sometimes 25% of the highest rate. The most likely explanation for this is that the operating system randomly allocates shares of timeslices to the individual threads in each process, and since all threads would be running at full pelt, they each get all of their timeslice shares. The easiest way to work around this problem was to use a very large sample size for averaging. Thus for each benchmark, 100 samples were taken and averaged.

The results for the request-reply benchmark are shown in Figure 9.1. Unusually, the uniprocessor machine got better performance than any of the multiprocessor machines, even in single-core mode. One possible explanation is that a multiprocessor machine has an inherently higher cost of I/O, perhaps due to an I/O bus or mutex that must be locked for I/O to take place – such an operation is more expensive for a multiprocessor machine as it requires cooperation from all processors.
9.2. PERFORMANCE EVALUATION

Why is the single-core mode slower for machine 2, but faster for machine 3? The intuitive expectation is for the single-core mode to be faster for this I/O-bound program. However if I/O is fast enough on machine 2 such that context switching is more expensive than I/O operations between processors, we can expect that the multi-core mode would be faster. This is the most likely explanation for machine 2’s behaviour.

The results for the publish/subscribe broker benchmark are shown in Figure 9.2. The higher speed of the uniprocessor machine is for the same reason as before, and the increased speed of the multicore mode can be explained by the more CPU-bound nature of the publish/subscribe example.
Figure 9.2: Publish/subscribe broker benchmark results
Chapter 10

Conclusions and Future Work

To conclude, we will revisit each of our major chapters and make sure that we provide evidence to support the contributions we claim in Section 1.1.

Our design in Chapter 3 implements the majority of Ninja primitives and constructs. We can thus say that our design adapts Ninja to C++. We may even claim that our design goes further than Ninja, since it introduces a number of constructs, such as the dynamic channel and session forking.

We have an implementation of our design, as described in Chapter 4. It works, and has been used successfully with participants that conform to our design in Chapter 3.

Our delegate extension to Ninja as described in Chapter 5 is sound and integrates well with the Ninja design. Delegates created in our implementation according to the implementation notes given in the same chapter are sound and work well.

Our publish/subscribe protocol given in Chapter 6 is well specified, its implementation conforms to the design we give in the chapter, it works and has been well tested. Similarly for the election protocol.

Our linearity check as described in Chapter 7 is based on well-known program analysis principles, its implementation has been tested with most of the example programs included with our implementation.

Our type inference mechanism as described in Chapter 8 is based on a carefully specified unification technique, and it has been used to retype many small examples included with our implementation producing the correct type.

We will now consider a number of interesting avenues we may pursue that are opened up to us by our work on this project.
10.1 Linearity Checks

This section shall describe some possible applications of the linearity constraint outside of the context of our language, but in the context of imperative languages.

One interesting application is to enforce through the type system the constraint that we will not carry out a given operation $x$. Suppose we (perhaps implicitly) pass to every procedure an object of linear type $t$, which is capable of action $x$ and is the only way of doing $x$. Every procedure must also return a value of type $t$. The only way this can happen is if the procedure does not read the object, but simply passes it to other procedures either when it calls them or via its return value.

Another application in the same vein is in usage-restricted objects. Consider the linear type $t_k$ which is capable of performing $x$ once, and which returns another linear object of type $t_{k-1}$. We can restrict the number of times $x$ happens to $n$ via the type system simply by passing in an object of type $t_n$ to begin with. We can also determine statically how many times an object does $x$ simply by subtracting the subscript of its return type $t_?$ and its parameter type $t_?$.

10.2 Alternative UNIX Implementations

There is another implementation of our communication system under UNIX that we could have considered. Instead of working with UNIX sockets and I/O primitives, communication could occur over a System V shared memory space, using mutexes and other threading primitives to control access to the memory space. This methodology was not considered, the main reason being time. Under an operating system such as Linux, it is likely that the I/O primitives provided by the operating system would be as fast or faster than any implementation that we could muster up on our own, simply because the Linux developers have had a much longer time to think about intercore I/O than I have, and thus would probably have implemented a safer, faster solution. Furthermore the kernel has a more direct level of access to the hardware than I would have with my shared memory and threading primitives, so there were probably some optimisation tricks that the kernel developers took advantage of that I am not privy to. Therefore it would not have been worth it to implement our own communication protocol, at least for a Linux implementation. That is not to say that other operating systems would not benefit from a more direct approach. Once a direct approach has been implemented, benchmarks can be run to determine the most appropriate choice.

10.3 Multicore Implementation

Although our current implementation runs on multicore CPUs such as the Xeon, it does so only via the multiprocessing features provided by the operating sys-
tem. Some multicore designs, such as the Cell Broadband Architecture, provide their own intercore communication mechanism independent of the operating system. An extension to this project would adapt our implementation to the communication mechanism of one of these architectures. Such communication systems are usually radically different from anything the operating system would provide, so a major part of the adaptation would be the mapping between our communication primitives and the target platform’s.

10.4 Catching Algebraic Type Misuses/Constructor Modifiers

As we noted in Section 4.6, we sacrificed type safety in the name of convenience when we allowed an algebraic type with constant parameters be passed to the receive method. It would be easy to catch this error with a specially constructed ROSE program traversal, but perhaps a more general solution is possible. At present, constructors cannot hold modifiers to modify the type of the class that they construct. But this is because the only two modifiers are const and volatile, both would not make sense for a constructor. But consider if the attribute pnonconst were attached to objects created using the non-const algebraic type constructor, and the attribute were required by the parameter to receive. Now the situation we discussed is no longer possible. This is very preliminary and a lot of work would need to be done to determine type conversion semantics of modifiers such as this, but it shows promise.

10.5 The pwhile and pfor Loops

One statement mentioned in the Ninja specification but unimplemented in our language is the pwhile loop. The goal of this loop is to provide a means for two loops to stay synchronised – as soon as the loop condition for the ‘master’ loop returns false to terminate the loop, a message to this effect is sent to the other participant, which will terminate its own loop. A similar construct not mentioned in the paper is the pfor loop – the number of iterations for a pfor loop is computed before the loop begins and is transmitted once, thus avoiding a message for every iteration. The messages themselves are not particularly hard to implement – for example, we could have 0-ary PwhileContinue and PwhileBreak algebraic types, and a unary PforCount algebraic type with an int parameter. The trick is integrating the message into the session type in a seamless way and – for the pfor loop – providing a way of exiting the loop after the desired number of iterations without sending any extraneous messages or breaking our type system.
10.6 Sessionifying Existing Programs

Many programs use existing untyped IPC mechanisms such as Unix sockets or System V shared memory. An interesting extension to this project would implement a tool that converts such programs into a rudimentary untyped participant. The untyped participant can then be typed using our inference system. Using the type information we can show correctness, compatibility, or perhaps some other property.

10.7 Handling Error Cases

Our implementation currently does not perform any error handling at all. This is because the effects of an error case on the session type have not been well studied. What happens when a communication failure occurs, or a participant unexpectedly terminates for any number of reasons? Clearly the peer’s session type is no longer valid, but how should it be reflected in the implementation? What about error conditions that we can trap and safely continue or retry? What would be the best representation of these conditions in the session type, or should they not even be reflected in the session type at all?

10.8 Sessionifying Plaintext Protocols

There are many commonly used plaintext protocols, such as SMTP, that exhibit stateful behaviour. It is possible to characterise the behaviour of these protocols using a session type, however since our wire protocol does not necessarily correspond directly to a plaintext protocol, we cannot use the wire protocol directly. We can however implement a translation participant which handles the plaintext parts of the communication on behalf of the implementor participant, in that it presents commands and other messages received on the wire as data in appropriate algebraic types to the implementor, and forwards communication from the implementor onto the wire formatted as a plaintext message. Provided that the translator works correctly, and the session type is correctly specified, we can be assured of the correctness of the implementation with regard to the protocol specification.

10.9 Multi-Channel Communication Procedures

It should be possible to construct a communication procedure that listens on more than one channel. This would be reflected in the prototype of the communication procedure; we would expect it to have as many session parameters as it has channels to listen on. This feature would be useful for participants which need to draw, or act upon, information from a large number of non-homogeneous sources.
There will be an interval of time between when the first channel is invoked and when the last one is invoked. For some communication procedures to work effectively, they should be able to communicate on all channels as soon as it begins to run. For others, they need to start working as soon as possible. This imbues upon us two possible designs for the interface between the participant and the communication procedure. If the participant starts the communication procedure as early as possible, the procedure needs to be aware that some communication primitives may block or be otherwise unavailable. It would also be worthwhile to provide a mechanism for determining if a given channel has been invoked yet.

If the participant starts the communication procedure after all channels have been invoked, the extra element of blocking would be imposed upon the invoker. Either the invocation step would block (this does not currently happen under normal circumstances), or the first communication may block, until the communication procedure’s channels have all been invoked.

10.10 Participant Behavioural Restrictions

In the same way as we currently restrict the communication behaviour of a session through the session typing mechanism, we may also wish to restrict the behaviour of a set of sessions, or perhaps an entire participant. The main benefit of imposing such restrictions is that we lift the specification capabilities of the session typing mechanism from a channel-wide level to a system-wide level. That is, we can impose restrictions on the system as a whole, rather than on a single channel. This has clear benefits, especially in the context of software components which as a whole must conform to an externally provided specification.

The restrictions we impose are in addition to the session typing restrictions we already impose. One technique for imposing such requirements is given here. We shall construct a typed session tuple, which is a tuple of sessions, each with their own type. In addition to the session’s types, the typed session tuple is parameterised over a session tuple type. A session tuple type is similar to a session type, except that actions have an additional parameter: the session index. This is an index into our session tuple which specifies the only session which may perform the given action. For a communication action to take place, it must conform to both the session tuple type as well as the session type of the session at the specified index (all communication actions will give an index into the tuple). Similarly to a session, a session tuple will mutate its session tuple type to reflect any communication actions that occur, as well as the appropriate session type itself.

To impose restrictions on all sessions in a participant, we need to add all the sessions used by the participant to our tuple, and use the tuple for any communication. This technique will only work if the participant uses a fixed number of sessions.
10.11 Delegation to Participants

[17] gives a second definition of delegation: “the sending and receiving of sessions over sessions”. How might we implement this in our language? At the high level, there are two things we need to take into consideration when this happens. The first is the registration of the session type. As with all types sent over a channel, the session type must be registered. This may pose a problem for us if the session type is parameterised. The second is the linearity constraint. Since the session we send over the session is accessed at the time of sending, both the aforementioned sessions become invalid (as they should do).

At the operating system level, what we would like to do is send a filehandle over a UNIX socket. Luckily, UNIX provides us with a way of doing this. There is an \textit{ancillary data} mechanism defined in POSIX.1g which allows for transmission of certain out-of-band data, including file handles. To send a session, we simply need to send the filehandle as ancillary data, transmit a notification (which may simply be the session’s type identifier) which will trigger the receiver to pick up the filehandle.

For other operating system implementations, things might get harder, say if there is no way to send filehandles at all, or easier, say if we simply need to pass the address of a shared memory area,
Bibliography


[26] Todd L. Veldhuizen. C++ templates are Turing complete.


