Type Representations and Coordination

Andrew Wilkinson

A Thesis Submitted for the Degree of PhD

University of York

Department of Computer Science

May 2007
Abstract

Open coordination systems are a means of performing distributed computing where the processes in the system are not known to each other until the system is run. In an ideal world these processes could be of any language and running on any hardware or operating system. Traditionally systems have limited processes either to a single language or so that they can only communicate using simple types.

In this thesis the notion of what makes two types equivalent, including differences in the representation as binary values of seemingly identical types and differences caused by representing types in different languages, is discussed. Type isomorphisms are used as a method of both determining the equivalence of two arbitrary types and of generating the functions needed to convert values between values of those types. This work is placed into the context of open coordination systems, with special focus on LINDA.

A method of efficiently using type isomorphisms as the type matching algorithms in a LINDA-like system is developed and evaluated. Allowing complex types to be used imposes some additional requirements on the library that client programs use to connect to a LINDA network. Two methods that the client library could use are discussed, compared and evaluated.
Contents

Acknowledgements 11

Declaration 13

1 Introduction 15
  1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  1.2 LINDA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
  1.3 Types In Other Systems . . . . . . . . . . . . . . . . . . . . . . . . 18
  1.4 Advantages Of LINDA . . . . . . . . . . . . . . . . . . . . . . . . 19
  1.5 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  1.6 Language Differences . . . . . . . . . . . . . . . . . . . . . . . . 21
  1.7 Hardware Architectures . . . . . . . . . . . . . . . . . . . . . . . . 23
  1.8 Networking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
  1.9 Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
    1.9.1 Atomic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
    1.9.2 Product . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
    1.9.3 Sum . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
    1.9.4 Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
  1.10 Representation of Types . . . . . . . . . . . . . . . . . . . . . . . . 29
  1.11 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
  1.12 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

2 Literature Review 33
  2.1 Linda . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
    2.1.1 Open Coordination . . . . . . . . . . . . . . . . . . . . . . . . 33
    2.1.2 Tuplespaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
    2.1.3 Extensions to Linda . . . . . . . . . . . . . . . . . . . . . . . 36
    2.1.4 Multiple Tuple Spaces . . . . . . . . . . . . . . . . . . . . . . 37
    2.1.5 Non-Blocking Primitives . . . . . . . . . . . . . . . . . . . . . 38
    2.1.6 Bulk Tuple Operations . . . . . . . . . . . . . . . . . . . . . . 38
    2.1.7 Garbage Collection . . . . . . . . . . . . . . . . . . . . . . . . 40
    2.1.8 Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
    2.1.9 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
  2.2 COM and DCOM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.3 Unregistered Type Information ........................................ 129
6.4 Registered Type Information ........................................ 133
6.5 Comparison .......................................................... 137
6.6 Optimisations .......................................................... 139
   6.6.1 Preemptive Isomorphism Calculation .......................... 141
   6.6.2 Garbage Collection .............................................. 143
6.7 Implementation .......................................................... 144
   6.7.1 Unregistered ....................................................... 144
   6.7.2 Registered ......................................................... 147
6.8 Evaluation Of Changes .................................................. 150
6.9 Conclusion .............................................................. 150

7 Case Studies ............................................................... 153
7.1 Prime Sieve .............................................................. 153
   7.1.1 Implementation .................................................. 155
   7.1.2 Conclusion ......................................................... 156
7.2 Sudoku Solver ............................................................ 156
   7.2.1 Without Compound Types ...................................... 159
   7.2.2 Comparisons ....................................................... 159
   7.2.3 Conclusion ......................................................... 161
7.3 Sudoku Generator ........................................................ 161
   7.3.1 Testing Grids ...................................................... 162
   7.3.2 Generating Grids .................................................. 162
   7.3.3 Types ............................................................... 163
   7.3.4 Non Isomorphic Types .......................................... 163
   7.3.5 Conclusion ......................................................... 163
7.4 Triangles ................................................................. 164
   7.4.1 Slave ............................................................... 164
   7.4.2 An Additional Process ........................................... 164
   7.4.3 Conclusion ......................................................... 165
7.5 Implementing eval ........................................................ 165
   7.5.1 Implementation .................................................. 166
   7.5.2 Conclusion ......................................................... 166
7.6 Conclusion .............................................................. 167

8 Results ................................................................. 171
8.1 Testing Environment ..................................................... 172
8.2 Measureables ........................................................... 173
8.3 Outputting Tuples ........................................................ 175
   8.3.1 Complex Types .................................................. 177
   8.3.2 Memory Usage .................................................... 178
   8.3.3 Network Usage .................................................... 179
8.4 Retrieving Tuples ........................................................ 180
### C Getting And Running The PyLinda Code

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1 Getting And Building PyLinda</td>
<td>225</td>
</tr>
<tr>
<td>C.2 Running PyLinda</td>
<td>226</td>
</tr>
<tr>
<td>C.3 Problems or Bugs</td>
<td>226</td>
</tr>
</tbody>
</table>

### D Case Study Code

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1 Prime Sieve</td>
<td>227</td>
</tr>
<tr>
<td>D.1.1 Master</td>
<td>227</td>
</tr>
<tr>
<td>D.1.2 Slave</td>
<td>228</td>
</tr>
<tr>
<td>D.1.3 Prime Sieve</td>
<td>228</td>
</tr>
<tr>
<td>D.2 Sudoku Solver</td>
<td>229</td>
</tr>
<tr>
<td>D.2.1 Traditional Version</td>
<td>229</td>
</tr>
<tr>
<td>D.2.2 Compound Types</td>
<td>232</td>
</tr>
<tr>
<td>D.3 Sudoku Generator</td>
<td>235</td>
</tr>
<tr>
<td>D.3.1 Master Process</td>
<td>235</td>
</tr>
<tr>
<td>D.3.2 Generator Process</td>
<td>236</td>
</tr>
<tr>
<td>D.3.3 Solver Process</td>
<td>238</td>
</tr>
<tr>
<td>D.3.4 Difficulty Function</td>
<td>239</td>
</tr>
<tr>
<td>References</td>
<td>250</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 Two different implementations of multiple tuplespaces ........ 37  
2.2 Example of Using A Bulk Tuple Primitive .......................... 39  
2.3 The CORBA architecture ........................................... 44  

3.1 BNF Grammar For The Type Specification Language ............... 77  

5.1 Idealised C code for communicating with LINDA ...................... 112  
5.2 Idealised C code for communicating with LINDA ...................... 112  
5.3 Haskell code embedded into Python ................................. 117  
5.4 Haskell code embedded into C ...................................... 117  
5.5 The Fibonacci sequence implemented in MINIMAL inside C .......... 122  
5.6 The Fibonacci sequence implemented in MINIMAL inside Python .... 122  

6.1 Using A Nameless Type Registry ..................................... 138  
6.2 The XML message used to place two tuples into a tuplespace ...... 148  

7.1 The master process .................................................... 155  
7.2 The slave control process ............................................ 156  
7.3 The prime sieve process ............................................. 157  
7.4 A Sudoku Grid in C .................................................... 158  
7.5 A Sudoku Grid in Python ............................................. 158  
7.6 Traditional version of the Master process ............................ 160  
7.7 Typed version of the Master process ................................ 160  
7.8 Type Of The Difficulty Function .................................. 162  
7.9 The Slave Process ..................................................... 164  
7.10 Using MINIMAL to complete the triangle ............................ 165  
7.11 Using C to complete the triangle ................................... 168  
7.12 An implementation of eval in C ................................... 169  
7.13 The eval monitor code in C ........................................ 169  
7.14 A simple example of the use of eval in Python ...................... 169  
7.15 Creating multiple tuples from a single eval in Python ............ 170  

8.1 Code For The Out Tests .............................................. 176  
8.2 Time Taken To Output 10,000 Tuples ................................ 176  
8.3 Code For The Compound Out Tests .................................. 177  
8.4 Time Taken To Output 10,000 Tuples With Compound Types ...... 178
Acknowledgements

Making it through to the end of my PhD studies alive is not a task that I could have achieved alone. I wish to thank the people who helped me in many ways through the three and a half years it has taken.

My studies have placed a large financial burden on my parents, but it is a burden they have accepted without complaint and still maintained a level of enthusiasm and encouragement that is bafflingly high. It is impossible to put how much I owe them into words. Chris, my brother, has also been a great distraction and a great friend. Without our long discussions about the (mis)fortunes of Peterborough United I would have surely lost touch with reality a long time ago.

Dr Alan Wood has been a constant help throughout my work and without his guidance this thesis would surely have never been written. From my early fumblings to the final polishing without his assistance my PhD would have remained nothing but a pipe dream.

I would also like to thank all the members of the Plasma group for their suggestions regarding my own work, and their help in broadening my own knowledge of Computer Science through their talks and informal discussions. I would especially like to thank Nur Izura Udzir for her help with many aspects of my work, and for providing impetus for me to complete PyLinda. Neil Mitchell and Tom Shackell also deserve a special mention for providing excellent lunch companions.

Finally I would like to thank Ellen Adams for her patience and understanding as the many deadlines I set myself went whizzing past. I look forward to buying her some shoes as thanks.
Declaration

I hereby declare that, unless otherwise indicated in the text, the research presented in this thesis is original work, undertaken by myself, Andrew Wilkinson, between October 2003 and April 2008. I have acknowledged external sources through bibliographic referencing.

A general overview of the work presented in this thesis was given in [1]. The major contributions to this paper were made by myself with input from my supervisor, Alan Wood.
Chapter 1

Introduction

Open coordination systems [2] such as LINDA [3] (see section 2.1) are conceptually simple methods for programmers to use distributed systems. The heterogeneous nature of both the hardware and software platforms that can be in use as part a distributed system lead to a number of type and type representation difficulties that will be discussed and resolved in this thesis. In open coordination systems the types that will be used by the processes which make up the system are not known until runtime. In addition differences in the underlying hardware can alter the representation of otherwise identical types.

Types are a fundamental way of retrieving values from a LINDA tuplespace [4]. For a LINDA-like system to be type safe the equivalence between types must be determined in order to retrieve values, and for types which are not identical the stored value must be converted into the equivalent representation for use in the process which retrieved the value.

1.1 Background

Since the first computers were created people have been using them to process larger and larger amounts of data. For tasks such as simulating the early moments of the universe, studying the explosion of an atomic bomb or predicting the weather, the amount of data required for an accurate model is enormous. Even for smaller tasks the time taken by an average computer to complete the computation may be too large to be practical. Many of these large problems can be divided into several stages and portions of the input data processed in parallel. Even in 1943 the benefits of having multiple computers solving different portions of the same problem were recognised and by the end of the second World War nine Mark II Colossus machines were running in parallel at Bletchley Park decoding encrypted German messages.

The scale of the problems tackled by computers has grown at an enormous rate. Among the most well known users of distributed systems today is Google, whose huge warehouses full of computers scour the web and service search requests from millions of users a day. Tackling problems involving the size of data used by Google and others would be completely impossible without a large array of computers working
together to solve a common problem.

In the early days of computer science programmers worked directly with the machines, programming everything in machine code - the actual instruction language executed by the processor. This style of programming was difficult and error prone so languages were designed that made it easier for a programmer to write correct code. Over time a huge range of languages have been developed, from languages that expose much of the underlying hardware architecture like C to languages that insulate the programmer from those details, such as Haskell [5] and Python [6]. Each language has its own strengths and weaknesses and every programmer has their own language preferences. There is no ‘perfect’ language that fits every possible programming problem and instead different languages are chosen because they are a better fit for the different problems being tackled.

One of the few common elements among all programming languages is that there is a translation step where the code written by the programmer is either converted into a form executable by the host processor or it is translated as it runs by a virtual machine running on top of the host machine. Some languages translate the code into a form that can be interpreted more easily, but isn’t directly executable by a processor. As part of this process values that are used in the program are converted into a string of binary digits that can be placed into the computer’s memory. The conversion between the types as they are defined in the code and how they are represented in the running process’ memory can be a complicated process. While a programmer may use the type ‘integer’ this may equate to several different representations of that type, depending on the compiler used and architecture is being compiled for. In cases where the code is being compiled to run directly on the host machine other issues such as memory alignment may also affect the representation.

Machine code consists of a series of instructions which usually take one or more arguments. These arguments refer either to the registers of the processor or to a location in the machine’s main memory. Although the processor can perform arithmetic operations on these elements they only have a weak notion of type. The binary representation of an integer will also be a valid representation of a floating point number, providing they are the same length\(^1\). If the binary representation of the integer ‘1’ is interpreted as a floating point value then it becomes ‘5.296829’ not the expected ‘1.0’. It is quite possible to write an integer value to a memory location and then to operate on it as if it were a floating point value. The processor will not hesitate to perform these operations: operations that in all sane cases should be disallowed. For this reason a computer’s memory can be considered to be untyped. The untyped nature of a computer’s memory makes the life of a machine code programmer very difficult. There is so little error checking in the language they are using that it is easy to make errors such as adding a float as if it were an integer.

With the development of higher level languages came the notion of typed vari-

---

\(^1\)And assuming that all possible values for a binary value of \(n\)-bits is a valid value for the integer and floating point types. This is the case for 2s complement integers and the IEEE floating point standard, the two most common formats in use.
ables. These are elements of memory where the compiler will guarantee that it will only be operated on in the correct manner. If the programmer attempts to add a float as if it were an integer then the compiler would detect an error before the program is even run. These languages allow the programmer to create their own types which combine a set of base types provided by the compiler to form more complicated types, which are still guaranteed to be operated on correctly.

When programmers have a large problem to solve, a problem too large for a single machine, they may choose to spread the work over a group, or cluster, of machines. How the work is spread across the machines is vital to the success of solving the problem. If the work is spread inefficiently then the performance of the system may be worse than running it on a single machine! When work is distributed evenly between nodes in the cluster then it is possible to reap large performance gains. Huge projects such as Seti@Home\(^2\) would be impossible if it did not use many separate computers to perform the work. When Seti@Home was launched in 1999 it took less than a year to log over two million years of computing time [7] using more than 900,000 computers spread around the world. Although it may be possible to calculate results faster by using a localised cluster of machines rather than have volunteers from across the world, the performance to cost ratio is far better.

Some programming problems naturally lend themselves to parallelisation. Problems that consist of a number of pieces of work which can be processed at least partially independently are ideal. Trying to implement such a problem in traditional, serial programming languages would result in the programmer being forced to make decisions that are irrelevant to the problem, such as what order should the pieces be processed. By using a distributed computing platform the programmer can concentrate solely on the problem.

1.2 LINDA

One method of coordinating the work of several processes in a distributed environment is LINDA [2, 3]. LINDA is a tuplespace-based open coordination system with an extremely simple and flexible set of primitives. The central notion underlying LINDA is that of a tuplespace. A tuple is an ordered, typed chunk of data while a tuplespace is a bag (a multi-set) of tuples. From the client's perspective LINDA consists of three core operations, \texttt{in}, \texttt{rd} and \texttt{out}. The \texttt{out} primitive places a tuple into a tuplespace while the \texttt{in} and \texttt{rd} operations retrieve one. A full description of the LINDA model is given in section 2.1.

The three LINDA primitives are language agnostic and can be added to any programming language to provide coordination facilities. One of the driving principles behind the design of LINDA was that coordination is orthogonal to computation. What this means is that the features of a programming language that allow it to compute results are entirely separate from features that allow communication be-

\(^2\)http://setiathome.berkeley.edu
tween processes. This results in LINDA being used as a library by the programmer. It is an extension to their chosen language which provides additional functionality that they can use.

The primitives `in` and `rd` both take a template and return a tuple that matches. The matching is done using the types of the elements from both tuples. A template is a tuple which can contain values (with their associated types) as elements, or types. For the template to match a tuple they must have the same number of elements and each of the elements must match. If the template element is a value then it must be of an equivalent type to that in the tuple and they must have the same value. If the element in the template is a type then the tuple element will match providing its type is equivalent to the template element.

The handling of types is clearly central to a successful LINDA implementation. However, previous implementations of LINDA take one of two routes, either they use a single language and rely on the built in functionality for serialisation and type matching or they restrict the user to a small fixed set of types. Implementations that choose to use a language such as Java [8] as the sole language for writing client applications can take advantage of the fact that the language provides many, if not all, of the features for handling types. However, this leads them to ignore this important aspect of LINDA as type matching is provided for them. A LINDA implementation which restricts the user to a fixed set of types causes the LINDA primitives to be incongruous to the rest of the language. They are no longer completely orthogonal to the host language as the programmer will need to do some extra work to break down any compound types into suitable types for inclusion in a tuple. As well as causing the programmer extra work this causes problems as the system will no longer be type safe. If a programmer wishes to ‘out’ a pair of integers they must first split the pair into its constituent parts and output a two element tuple. A tuple containing a single element, a pair of integers, is clearly different from a two element tuple containing two integers yet this is what the system would force the programmer to use.

1.3 Types In Other Systems

Types are not just a problem for LINDA but all distributed systems. Whatever the architecture of a system it will involve moving pieces of data from machine to machine. These pieces of data will have types and will be represented in a specific format. If the machines have different architectures, or the data is being sent between different languages, then questions arise about the handling of the data. If all the different representations that can ever be used in a system are known then it is relatively straightforward to have all possible conversions between the different formats generated and ready to use. What these conversions are, and how to generate them is a very similar problem to that which is being tackled in this thesis except it is decided at compilation time and this work is considering it at runtime. Solving these issues at compile time removes some of the difficulties, such
as the performance of the algorithms used to generate the conversion, but others like the specification of the different type representations and the actual generation of the conversions remain the same. The application of the algorithms, developed in this thesis, to other systems will be investigated in chapter 9.

While LINDA has proven to be a useful tool in studying coordination and distributed computing in general, systems such as Common Object Request-Broker Architecture [9] (CORBA) and Distributed Component Object Model [10] (DCOM) have become the de-facto standards in industry either by being included as standard in Windows (as is the case for DCOM) or by having a large industry consortium promoting it (as is the case with Object Management Group (OMG) promoting CORBA). Neither of these systems is an open coordination system, but they will be compared to LINDA to provide a background on other approaches to the handling of types in distributed systems.

Microsoft’s .Net architecture’s Remoting technology has in many cases superseded DCOM as DCOM was designed before the Internet became a priority for distributed systems. DCOM is still a useful and well studied system so it will be considered, along with the more modern .Net Remoting.

The most widely known implementation of LINDA is Sun’s JavaSpaces. Originally released as a commercial product, the software was donated to the open source community in 1999 as part of the Jini Technology package. There have been many other implementations of LINDA, some of which will be described in section 2.1, but few, if any, have progressed beyond purely academic interest.

The enthusiasm that has been shown for research on LINDA is due to the extreme simplicity of the idea that underpins LINDA and the great flexibility in its implementation. Unlike CORBA and DCOM a LINDA system can be created without complex architecture, as its specification does not require features such as fault tolerance, nor is there a large amount of work that needs to be done before a language can be integrated into the system. With CORBA, an Interface Definition Language (IDL) to host language compiler must be written, in addition to a library that supports all the complicated CORBA features, such as the Object Request Broker (ORB). To integrate LINDA a minimum of three simple functions can be implemented for a successful language binding.

1.4 Advantages Of LINDA

The extensive academic interest that there has been regarding LINDA means that there are a large number of variations to the LINDA model. This flexibility in implementation and the ability to focus on a small core and extend that, coupled with it being an open coordination system, mean that LINDA is a perfect implementation platform for the work proposed in this thesis. The openness of the system brings to the fore some type issues that can be worked around in closed system with the use of third-party type languages which are converted into the language used by the programmer. While common these solutions are ugly and the fact that they
are impossible to implement in an open system mean it is easier to focus on novel approaches.

LINDA’s explicit type matching makes it clear how types are involved in the system. Other distributed computing systems that use direct connections between two processes instead of the intermediate type-based storage of the tuplespace, still require type matching to take place. A message that contains any value, as most messages are likely to do, will, quite obviously, contain typed values. A process that is receiving a value as part of a message will be expecting it to arrive in a certain format and to be of a certain type. If this assumption is not checked then it is possible for the data to be in the wrong format and to be misinterpreted. This can lead to erroneous results or even crashes of the software.

Any changes made to LINDA must be compatible with other popular extensions that have been made to the core LINDA idea, such as multiple tuplespaces and the bulk tuple operations. Although the implementation of the work presented in this thesis will be done in a simple version of LINDA the possibilities for using the principles outlined in other systems, perhaps more suited to enterprise level applications, will be considered and discussed.

1.5 Definitions

The language used to describe coordination and distributed computing is often fairly vague and ill defined. The various pieces of terminology that will be used are defined below and it is these definitions that are intended throughout the remainder of the thesis.

Distributed computing is very broad topic covering all forms of computation which is performed across more than one machine. The classic example of distributed computing is Remote Procedure Calls (RPC). In recent times RPC has been replaced by Representational State Transfer [11] (REST) web-services that use Hyper-Text Transfer Protocol (HTTP) and Extensible Markup Language (XML) instead of a custom binary serialisation protocol to transfer function arguments. While RPC may be the simplest form of distributed computing the topic also covers more complicated systems such as CORBA and Grid computing [12].

In the seminal paper ‘Coordination Languages And Their Significance’ [2]格尔内特 and Carriero make a distinction between the coordination model and the coordination language:

The coordination model is the glue that binds separate activities into an ensemble. An ordinary computation language (e.g Fortran) embodies some computational model. A coordination language embodies a coordination model. It provides operations to create computational activities and to support communication among them.

They go on to describe how a coordination language can be added to a computational language to provide access to a coordination model. The concept of LINDA and other
coordination models is that they are embodied by a coordination language, which is orthogonal to the computational language. The line between a coordination model and a distributed computing model is unclear, but for the purposes of this thesis coordination models will be considered to be models that focus on how processes are connected, not what functionality they provide. One of the reasons that LINDA is an interesting model is that processes do not know the ‘name’ of any other processes and do not directly connect to others in the system.

Coordination systems can be further subdivided into two groups: open and closed systems. A closed system has all processes that will connect to the system specified before runtime. Closed systems are typically implemented using a special compiler that will handle all processes at the same time and use the knowledge that is gained from having all the source-code available to make optimisations. By contrast, in an open system each process that will connect to the network is compiled (and possibly developed) separately, so nothing can be inferred about the network until runtime.

1.6 Language Differences

At the most basic level every computer interprets a series of binary digits as commands which control the CPU. Writing software at this level is difficult and error prone, but it is possible to write programs directly in this way. The textual representation used when coding in this way, called assembly language, does not contain any more type information than is present in the underlying machine code. Any type information must be created, managed and enforced by the programmer.

Higher level languages, such as C [13], are mostly compiled down to the untyped machine code, but they do feature some type information. The C language is specifically designed so that it will prevent operations that are not permitted because of the type of the operands, but that the type system is flexible enough so that the programmer can override it. This allows the programmer to perform ‘dirty tricks’ such as taking a pointer to a memory location, which will have a type, casting it to an integer, discarding the type information in the process, performing some operations on the integer before casting the value back to a pointer, possibly even pointing to an object of a different type. This flexibility is due to C associating type information with variables not with the objects referenced by the variables. Objects are represented as a series of bytes in the computer’s memory with the rules that govern access to the objects enforced by the type of the variables that the objects are accessed through. Two variables may have different types but point to the same location in memory. Whatever actual type the object had is irrelevant and it will instead be treated as though it is the type of both variables that point to it. C’s union type encodes this behaviour as part of C’s type system.

Other languages borrow heavily in style from C and yet provide a more restrictive type system. Python is a popular example of such a language, along with Perl, Ruby, Ada and many others. An object-oriented, dynamically typed, imperative language, Python was developed by Guido van Rossum in 1990 to provide a scripting language
for the Amoeba distributed operating system. Since then Python has grown and evolved to the point where it can run on almost any hardware platform and under any operating system. Python programs run on top of a virtual machine that abstracts away from much of the details of the host machine. A Python program is not compiled directly to the machine code of the host and instead is compiled into a stack-based machine code that runs on the virtual machine. The virtual machine is not designed to run other languages and is tightly coupled to running Python programs.

The use of a virtual machine allows Python to view memory much more abstractly than C does. Rather than being a series of bytes, Python considers memory to be individual objects. These objects contain type information so objects cannot be ‘cast’ to another type as is the case with C. Although Python is strongly typed, it is not statically typed. Unlike in C variables do not contain type information and a variable in Python is simply a pointer to an object. More than one variable may point to the same object, and an object of any type can be pointed to from any variable. This allows the programmer great flexibility but with the safety of knowing that operations will be prevented if they are invalid according to the typing rules. The fact that variables are not typed and that typing errors are not detected until runtime mean that ‘duck typing’\(^3\) (the practice where the type of a value is not directly checked, and instead it is the methods provided by the value that determine if it is of the correct type) is an important part of Python programming.

Although C and Python share a number of features they have significantly different type systems. For this reason and to avoid complicating matters with discussions on the details of many other languages they will be the main two supported by the implementation. Other languages that will be mentioned in regards to future support by the implementation are introduced below.

Haskell is another high level language, but unlike Python it does not draw as much from C. Haskell is one of the purest functional languages in use today, and is itself a significant target for academic research. Haskell has a strict type system that statically infers type information throughout the program. As a pure functional language, Haskell does not have the concept of assigning to a variable in the same way that C and Python do. However, type errors can (and do) occur when passing arguments to functions and returning values. Haskell is a lazy language which allows it to have native support for some unusual types, such as an infinite list.

Scheme [14] is a language that is similar to both Python and Haskell. It is a functional language, like Haskell, although it is not as pure and has support for non-pure language elements such as assignment. Unlike Haskell, Scheme is an untyped language. Scheme has no support for building complex types such as classes although they can be implemented using a well known pattern involving nested functions. Scheme variants, and languages which are similar to Scheme do feature object orientation, such as Lisp’s Common Lisp Object System [15] (CLOS). How-

\(^3\)From the phrase ‘If it walks like a duck and quacks like a duck, I would call it a duck.’ attributed to James Whitcomb Riley
ever, discussion will be limited to pure Scheme to keep it as separate as possible from other languages discussed. Scheme’s almost complete lack of a type system make it an interesting language to study because it is so basic. Scheme’s minimal syntax and lack of an obvious type system will provide an interesting challenge for the system developed in this thesis. Haskell, Python and C all have an obvious type system that contains specific support for creating compound types in the language. Scheme’s differences in this respect make it an excellent language to study.

As has been described there are many languages each with their own unique concept of ‘type’. If these languages are considered individually then the type system that is used is relatively unimportant, the programmer will just use what is there. When considering transferring values between languages, whether across a network or simply through a file, the differences in the type systems are key to making it work.

1.7 Hardware Architectures

Computer hardware is just as diverse as the languages that run on top of it. For the purposes of this discussion hardware can be divided into groups based on two factors, “endianness” and the word size of the processor. In chapter 5 the different machine code languages used will become important, but it will be shown that the differences are too great for values to be converted between different languages in a practical fashion.

For machines that have multi-byte words, the order in which these bytes are stored is important. If the byte with the lowest address represents the lowest value of all the bytes then the machine is said to be little-endian, if the first byte is the highest then it is big-endian. The differences in byte orders are not tremendously significant, even if you are transferring binary data between machines of different endianness. The differences in endianness are caused mostly by differences in the circuit design of the CPU and probably personal preference on the part of the designer. There is no significant difference between the two choices in terms of speed or any other metric. The problem of different endianness first occurred when early computer networks were developed. The TCP/IP standard [16] is defined such that all multi-byte values that are part of the specification are in big-endian format, thus the big-endian ordering is also known as network order.

The word size of a processor typically refers to the size of the processor’s address bus or the size of the data bus. This is the processor’s standard size of integer, and is probably the most commonly used data type on that platform. This size has increased over the years, starting from 16 bits in the original Intel 8086 design, to the 64-bit processors that are available today. Other, less common processors have used more unusual values such as 12, 18, 24, 36, 39, 48, and 60 bits.

The word size of a processor is often determined by the size of the value that is used to hold a character. Before the 1960s a character was held in 6 bits, limiting the number of characters to 64, which meant that word sizes were typically multiple
of sixes. When the IBM System/360 was introduced it used 8 bits for a character and this became the de-facto standard which continues to the current time. Some early machines, such as the IBM 702, 1401 and 1620, used a variable word length. In these machines a number has no specified size, instead the end was detected when a character with a special code was encountered. Rather than storing a number as a binary value they used binary coded decimal format, but this is not used in any modern machines.

The size of a word has a large effect on how the processor accesses the computer’s memory. Some architectures use a word as the resolution of the memory addressing while others use the byte. This limits the size of the smallest unit of data a processor can access. If the address resolution is a word on a 32-bit system then the processor can only access data in 4 byte chunks. The most common processors around today use the byte as the addressing unit, but their complex instruction sets also contain instructions that are optimised to work on data that is aligned on a multiple of a word boundary. On a 32-bit machine this may mean that a processor can add two numbers together whatever their address. However, if both addresses are exactly divisible by 32 (or 64, or 128 etc.) then a special instruction can be used that executes faster.

Differences in the word sizes, addressing resolution and address alignment show how types that may be identical in the program’s code can be compiled to an entirely different binary representation. The exact format is usually consistent across the same operating system and compiler (specified as an Application Binary Interface (ABI)) but different compilers for the same language may give the same type a different representation. In some systems, such as the x86_64 architecture (the 64-bit extension to the x86 architecture, as used in the AMD64 processors), the ABI is not consistent across the system. As x86_64 processors can run both 32-bit and 64-bit code concurrently, both 32-bit and 64-bit versions of shared libraries, which have different ABIs, have to be available.

Some languages, include as part of the specification, a virtual machine that programs must be run upon. The Java Virtual Machine [8] (JVM), which is an integral part of Java’s ‘write once run anywhere’ philosophy, is defined to be big endian, regardless of what the underlying hardware’s endianness is. To actually operate as a big endian machine, even when running on a little endian host machine, would introduce a large overhead. To compensate for this, implementations of the JVM internally use the host machine’s endianness and define operations that are affected by the byte order so that they make the representation appear to be big endian. This is an example of how programmers can be lied to by their compiler, and are tricked into thinking their data is represented one way when in fact it is stored differently.

As previously mentioned Python contains a virtual machine where the memory consists of objects rather than bytes or words. A Python integer object is internally implemented using a native integer. However, the integer object itself contains other
extra information used for things like garbage collection. As the byte codes in the
virtual machine operate on the objects, rather than bytes, it is impossible to interpret
the representation of one type as another. Java was designed to be implemented in
hardware, as well as a virtual machine, thus it was necessary to specify details such
as the byte ordering. As the Python virtual machine was only designed to run on
top of a host machine, and never to be implemented in hardware, integers can be
implemented using the native integer representation, whatever that may be.

The selection of hardware differences discussed in this section show that there is
a great breadth in computer hardware. The problem of converting, automatically,
between different representations of identical types, purely based on the underlying
hardware, is a difficult problem. It is not clear to programmers how their types
will be represented and, indeed, it should not be. In order to send a value to
a different machine the programmer will usually have to create custom serialisation/
deserialisation code that converts the value into a well specified format that is
agreed before the code is written. This is an error prone process that can prove very
difficult to debug. This is a world away from the ideal situation of being able to say
‘here is a value of type X’ and transmit it to another process. Languages that do
include a generic serialisation protocol often have some significant weaknesses. Both
Python and Java include modules in their standard libraries that do this. Python’s
Pickle [17] module requires identical code to be available at the destination, although
it does not check for this and this can cause serious problems if this requirement is
not met. Java on the other hand can transfer the required code, but this can involve
significant network traffic.

1.8 Networking

A key element of any distributed system is the networking capability of the machines
it is being run on. A computer network is a concept that may be physically imple-
dented in a large number of ways, or it may not exist outside of a single computer.
A single, stand-alone computer will have a networking capability which allows com-
munication between processes running on it. The key component of a computer
network is that it allows real-time communication between processes running on one
or more computers. A network may be as simple as hooking a null-modem cable
between two computers or as complicated as using a wireless broadband router to
connect a cluster of computers to the Internet. The exact physical connection that
carries the network traffic could be an 802.11b wireless connection, Ethernet cable
or even carrier pigeon [18, 19].

The physical characteristics of the network are irrelevant for the purposes of this
work, so a network will be considered using the socket metaphor that is implemented
in the widely used BSD network Application Programming Interface (API) library.
Although TCP/IP is only a part of a complete network stack, that stretches from
the wires that make up the physical network, to the highest level of the messaging
protocol, it is the level at which most programs that use networking work. Unless
it is being implemented on top of a ‘middleware’ layer that abstracts away from these details a process will communicate directly to another process using a socket. A socket represents the end of a pipe, down which a stream of bytes can be sent. Connections over a network are bi-directional so sockets actually represent two pipes, one which can be read from and another which can be written to.

Sockets provide no structure beyond the byte. There is no built-in method to know how large a message is or to provide a structure on the data being sent or received. Sockets also provide no built-in way to ensure that they have connected to a process that is expecting data in the format that the programmer is going to send. Socket connections over a network are initiated using an IP address and a port number. The IP address specifies which computer to connect to and the port specifies which process on that computer you wish to communicate with. Only one process may listen on a given port at any one time, but there is no restriction on which process may listen on which port and a single process may listen on more than one port at a time. Ports are simply an integer number between 0 and 65535, but ports between 0 and 1023 can only be used by processes run using the superuser, or ‘root’, account.

Due to the lack of any structure on the communication, processes that use sockets must implement their own. A number of protocols have become widely used over the years, the most prevalent of which is probably HTTP. HTTP was introduced by Sir Tim Berners-Lee [20, 21] when he created the World-Wide Web and is a simple, stateless protocol for transferring data between a client and a server. HTTP provides a simple method to access different parts of the server through the use of paths. These paths do not need to be physical locations on the server and can instead be entirely virtual. The path is simply a string, the value of which follows a few simple rules, such as a ‘/’ character representing the divider between different levels in a hierarchy. Although paths originally represented the physical file system on a machine, the server can treat a path in any way, including using it to refer to methods in the server software itself.

HTTP has become an almost ubiquitous way of communicating between servers. In most cases RPC is implemented using XML over an HTTP connection. XML is a standardised method of representing typed, structured data which will be discussed in detail in section 2.6.6. The free availability of well tested, secure and high performance server software and a multitude of technologies for implementing the ‘back end’ means that HTTP is an attractive proposition for implementing low level network connections.

Protocols such as HTTP only solve some of the problems that are faced when creating a distributed system. Although it does deal with issues such as connecting to a server and calling functions, which is easily implemented using virtual paths, it does not handle the contents of the message. HTTP is purely a wrapper around a message and the message is transferred untouched. The HTTP protocol does specify the sort of data being returned using Multipurpose Internet Mail Extensions [22] (MIME).
but this is a fairly broad description of the contents such as Hyper-Text Markup Language (HTML), XML or an image format (e.g. PNG). The MIME type is only useful if both ends of the communication have prior agreement on what the format represented by each code name is and how it is organised. This is precisely what this thesis is about, in this example the format of the message is agreed prior to runtime and the type is transferred purely as a name, a situation that is not possible in open coordination systems.

XML is a format that is often used with HTTP to form so called ‘web-services’, a form of RPC. XML is used to encode parameters while the Uniform Resource Locator (URL) specifies the function to call. The return value is encoded as XML and sent as the response to the HTTP request. Web-services can be used to implement distributed computing systems and so will be considered in more depth in section 2.6.7.

This section has given a brief overview of the low level networking mechanisms that under-pin all networked systems. Only a small number of technologies and protocols have been touched on in this section. HTTP is only one of a number of protocols, others such as File Transfer Protocol (FTP), Gopher or many other file transfer protocols could be used, but they all share the same problem of just wrapping an unstructured message. This thesis describes a method of giving structure to such a message, so that it can be received and understood by a process expecting an equivalent type.

1.9 Types

Previously the differences between various programming languages were discussed, as were the variations in hardware platforms. This section will focus on abstract types. Abstract types are used to separate type theory from the detail of any specific language implementations. As each language has its own unique concept of type, and many feature a wide range of built in compound types, the mapping between the actual types and the abstract types described may be non-trivial.

A common low level protocol needs to be used to allow even the most basic communication between two processes. In the case of the proposed system this involves defining a collection of types which can be used to describe the types used in actual processes. The representations of types are those defined by the programmer in each process. These are mapped on to the abstract types so that whatever language is being used they are talking in a common type language.

1.9.1 Atomic

The building blocks of all types are known as ‘atomic types’. Types that fall into this category include integers, floating point numbers and characters. These types are indivisible units that represent the smallest units of data it is possible to use in a program.
Representations of atomic types include the different sized and ordered integers, the various forms of floating point numbers and different encodings of character data.

1.9.2 Product

A product type consists of \( n \) types where each of the \( n \) types may be of the same or different types. In the C language the \texttt{struct} keyword defines a new product type, as does \texttt{class} in C++ and Python. A tuple is also a product type and although a tuple is indexed by position, while a \texttt{struct} is indexed by name, these are just different ways of projecting data from a value of the product type. The question of whether a product type is ordered (as in a tuple) or unordered (as in a \texttt{struct}) will be discussed in chapter 3.

1.9.3 Sum

A sum type allows a value to contain a single value of one of a number of other types. In dynamically typed languages all variables can be considered to be of a sum type, where the sum type consists of all the types available in the program. Technically every variable may be able to take a value of any type. It is unlikely for it to be actually used in this fashion and further analysis of the program may reveal that it is only ever assigned a subset of these types. The C \texttt{union} type is perhaps the most common example of a sum type as it allows values of a number of types to be treated as the same type. The significant difference between a \texttt{union} and a sum type is that once a value has been assigned to a \texttt{union}, the type information is lost. A sum type maintains this information and only allows the value contained in a value of a sum type to be extracted as this single type. A \texttt{union} value can have its binary representation interpreted as any of the component types.

1.9.4 Function

A third compound type is that of a function. In theory a function is a simple concept, it takes a number of arguments and returns a value created by some computation on those arguments. In reality few languages implement functions in such simplistic terms as most languages allow functions to have side-effects. Even Haskell, the purest language mentioned so far, allows functions that do not follow this definition through the use of Monads [23].

These four abstract types can be used to represent types that are defined in a programming language, and will allow different programming languages to compare type information. Later in this thesis how this type information is compared will be discussed and examples will be given for converting from concrete types to the abstract types. The reasoning behind having a common abstract type system will also be discussed in more depth.
1.10 Representation of Types

A key concept that will be mentioned time and again in this thesis is the idea of type representation. A type is a set of values with certain properties and that has a specific meaning. A type representation is a concrete implementation of this abstract idea. In traditional programming involving just one process, there is a single representation for each type. Some multi-process systems, such as CORBA, avoid the problem being tackled in this thesis by deciding on a common representation for types and generating the types that are used by the programmer from that. This forces the distributed system into having one representation for each type at the expense of making the programmer accept this single representation, whatever language they are using.

In the Linda model programmers write their programs in isolation. They create types in the programming language that they are using; types that are ideal for their particular program. When values of these types are placed into a tuplespace another process may try to retrieve them. The Linda matching algorithm is then a question of whether the type representation of the value, and the type representation being used to retrieve it are the same. Because we are considering types to have more than one possible representation the two type representations do not need to be identical, instead they only need to represent the same type.

Consider a collection of type representations. If these all represent the same type then there is a single object that all these representation derive from: the type. In the proposed Linda model of types this type is not specified. If it were to be, then the system would no longer be open as the type would need to be declared before the system is run. The rules that govern when type representations are considered to be representations of the same type, and the reasons behind them, are set out in chapter 4.

1.11 Implementation

The goals of this thesis have a very practical benefit, therefore a key element of the work is the implementation. As has been already stated, all Linda systems contain a type matching algorithm. Previously, this has consisted of either an equality test on a small set of atomic types, or was not specified and was instead provided by the language used to implement the system. The implementation work will focus on amending an existing Linda-like system, PyLinda [24]. The principal reason for choosing this system is that it is easily extensible and works well as a base for experiments involving Linda. In its initial implementation the PyLinda system contains no mechanisms for other languages to communicate with the system.

The PyLinda system currently features the multiple tuplespaces [25] and the bulk tuple operations [26] extension to the Linda core. A PyLinda system may contain many nodes and each tuplespace may be partitioned across one, many or all of these nodes. The system’s design was not focused on scalability or the overall
speed of the system but the alterations proposed will be checked to ensure they will not make the system any worse, and may in some cases improve performance.

The implementation work is divided into three sections, mirroring the theoretical presentation. Firstly, the type matching algorithm must be replaced. Secondly, the protocol for communicating between server and client must be extended to permit complex typed values to be transmitted with their structures intact. Finally, clients must be able to specify representations of types and transmit typed values using them.

Because the PyLinda server component is written in Python, a large portion of the code written, and thus the source code shown in this thesis, will be in Python. In order to allow multiple languages to communicate with the server it is necessary to write a language binding. A C library will be implemented that will allow a C program to communicate with the server. Where possible this library will be ‘wrapped’ in language specific code to provide the language binding for each other language tackled in this thesis. This will avoid the need to re-implement the low level socket handling code and instead focus on the integration with each individual language.

The process of wrapping a C library to provide functions in another language is a notoriously error prone and complex task. The details of the wrapping library for each language are irrelevant to the goals of the work and so will be omitted where possible. The focus for each language will instead be on the outcome of the wrapping, and how the language extension can be used within the context of that language and in the overall LINDA system. The only work this library performs is to serialise types and values so that they can be transferred across the network and implement the basic protocol for talking to the LINDA kernel.

For practical reasons the implementation will rely on free or open source tools and languages, especially those promoted by the GNU Project [27]. In many cases there are multiple implementations of the same language, Scheme being a particularly good example. When the language binding is introduced the reasons for choosing the particular flavour of the language will be outlined, but they will usually be openness, simplicity and ease of adding extensions written in C.

1.12 Thesis Outline

The thesis will begin with a discussion of the methods used in previous distributed systems to solve the type problem described above. This will focus largely on CORBA and DCOM as these are by far the most widely used systems for distributed computation. This discussion will also touch on other systems such as web services, XML and agent systems. Many of these systems rely on having the source code to all elements that can possibly be part of the running system, available at compile time. This is something that is absent from open coordination systems.

Chapter 3 will focus on the representation of types, the differences between hardware platforms and how different languages represent types in different ways. Chap-
ter 4 will build on this and will introduce a number of type isomorphism algorithms. These algorithms will be compared and their suitability discussed. Functions are a difficult topic and therefore chapter 5 will focus entirely on their use in open coordination systems. Chapter 6 will describe how type isomorphisms can be used in LINDA. In chapter 7 a number of case studies will be built to examine how complex type information can be used by programmers, and will show the advantages they have over a traditional, simple type system. Chapter 8 will analyse the implementation of the work in the thesis and discuss how the performance of the system has been changed. Finally chapter 9 will evaluate the work and discuss both the strengths and weaknesses of the proposed changes.

The discussion on type matching will show how a more complex algorithm can be integrated into an existing open coordination system replacing the previous basic type matching. As all LINDA systems must contain a type matching algorithm, even if it is undocumented or relies on existing language features, this replacement is relatively straightforward. Generating an isomorphism is likely to be more expensive than the algorithm it replaced. By analysis of the algorithm’s complexity, and empirical study of a running LINDA system, the effect this has will be shown. It will also be shown that through a few simple optimisation techniques the overall speed of the system is not significantly affected, nor is its scalability or fault tolerance.

Although the work presented in this thesis is designed to allow processes written in any language to communicate via LINDA it is not practical for the implementation to achieve this. Before a language can actually connect to the LINDA network a language binding must be written. This binding provides the functions that the programmer will call to perform the communication. Although the LINDA kernel is designed to be abstract enough to cope with any language, each language binding must be specially written.

A small but wide ranging selection of languages will be discussed and their language bindings described. The languages will be chosen to cover a broad range of imperative and functional languages, as well as statically typed, dynamically typed and untyped languages. It will be shown that creating a language binding that fits naturally into the language is a simple task and that the LINDA primitives and the new type functions are easy for a programmer to use, whatever language they are working in.

In addition to the complete language binding descriptions, a number of other, more esoteric, languages will have their language bindings discussed, in particular whether they are natural extensions to the host language. There are a vast number of available languages, far too many for even a fraction to be covered in this thesis. The languages that are covered have been carefully chosen to be representative of a much larger array of languages. With only minor changes from one of the discussed bindings any language could be successfully integrated into a LINDA-like system.

The technique of using type isomorphisms to declare two different representations of the same type as equivalent will be shown to be practical, but the utility of this
is harder to quantify. It is hoped that the combination of case studies and wide ranging language bindings will provide a strong argument that it is useful, practical and provides a new range of possibilities to a LINDA programmer.
Chapter 2

Literature Review

In the previous chapter the overall direction of this work was outlined. In this chapter, LINDA, the model that is used as the key example of an open coordination system will be introduced. Other distributed systems involving heterogeneous type systems will also be described. The distinction made between distributed systems and coordination systems is important, especially when open coordination systems are considered. In this chapter CORBA, .Net, XML and the like will be considered in the context for which they were designed. It will also be shown how this context is different from situations where an open coordination system may be used.

There are far too many distributed systems for them all to be introduced in detail, so only a few illustrative examples will be used. Most systems that are not mentioned can be sensibly compared to a system that is. For those that are described, a brief outline will be given for the systems general use and then a more detailed discussion of its method of solving type problems will follow. Each system’s approach to distributed computation will be compared to that of LINDA and the differences in the solutions to type problems discussed.

2.1 Linda

LINDA [2,3] is a coordination system which provides a simple interface for communication between processes. The central notion in LINDA is the tuplespace as it is through tuplespaces that all communication is done. Processes do not need to know the addresses or names of the other processes that are running in the system, they simply put messages into a tuplespace or wait for an appropriate message to appear.

2.1.1 Open Coordination

Coordination is a method by which several processes can collaborate to solve a common problem. The processes may be spread across a number of nodes in a network, or may be limited to a single machine. One of the important properties of a coordination system is that a process need not care about the physical location of another process in order to communicate with it. The word coordination also implies
a more collaborative method than the phrase “distributed computing”, although this is more a feature of how the systems are used than the systems themselves.

**LINDA** takes many of the concepts behind coordination and distributed computing to the extreme by completely detaching processes from each other. As well as a process not knowing the physical location of any others in the system, there is no method for direct communication between processes. All communication is performed through an intermediate storage medium, the tuplespace. The tuplespace provides a level of indirection between processes in the system and allows processes to communicate regardless of their physical location.

In the initial **LINDA** papers [2, 3] it was not described as an open system, instead the coordination was closed and conducted between a clique of processes known prior to runtime. Over time **LINDA** implementations [4, 28–33] changed to provide a more flexible and easy to use system. By requiring all the processes to be known at compile time it is possible to optimise tuple matching and placement as the processes that will produce and request certain types of tuples can be known at compile time. To perform this level of optimisation usually requires a modified compiler to be developed that scans a program’s source code looking for uses of **LINDA** primitives. Use of a modified compiler also allows the language being used to be modified to allow **LINDA** primitives to be fully integrated in a way that is not possible with a plain function call¹.

Optimisations to **LINDA**, such as automatically placing tuples physically near to a process that is likely to request them are hard to implement correctly. Creating (or at least modifying) a compiler to process the source code to allow these optimisations is also hard. There rather than focusing on fixed compile-time analysis, research interest turned to making **LINDA** a simple and light library that connects to a **LINDA** kernel where optimisations are performed dynamically as the system runs.

### 2.1.2 Tuplespaces

A tuplespace is a bag of tuples, used as the sole communication medium in **LINDA**-like systems. A tuplespace is a shared associative memory structure in that it is accessible to all processes, no matter their physical location. The definition of a tuplespace makes no formal requirement of the actual distributive nature of the implementation. All tuples in a tuplespace may be stored on a single node of the network, or they may be spread out over several nodes. Providing the tuplespace can be accessed directly or indirectly from any node on the network (and therefore appears as if it is shared) it does not matter where the tuples are stored. A tuplespace is an associative structure because tuples are accessed based on their value and structure, not by address.

As described in [3] tuplespaces give rise to a *generative* communication method.

---

¹Languages such as C do not treat types as first class objects. This makes it impossible to perform type-based tuple matching using the usual `int` to represent an integer as this, in C, is a syntax error. Languages that do treat types as first class objects can be cleanly implemented without using a modified compiler.
When a tuple is added to a tuplespace by a process that process may die but the tuple will remain in the tuplespace until it is requested by another process.

The main elements of a tuplespace based system are:

- **Tuplespace** This is a shared memory structure used to hold tuples. The early tuplespace based systems had only one tuplespace and allowed three operations on it - outputting a tuple, destructively inputting a tuple, and nondestructively reading a tuple.

- **Tuples** A tuple is an ordered, typed collection of values. The tuple \( \langle 1, \text{“Bob”} \rangle \) is a tuple of size 2, with an integer as the first element, and a string as the second.

- **Template** A template is a tuple, but with some or all elements replaced by a typed place holder that can be matched against tuples contained in the tuplespace. For example the template \( \langle 1, \text{string} \rangle \) will match \( \langle 1, \text{“Bob”} \rangle \), but not \( \langle 2, \text{“Bob”} \rangle \). A call to \texttt{in} will also not match \( \langle 1, 2 \rangle \) as the second element is not a string. In this example, the first element, 1, is known as an actual as it is an actual value while the second, \texttt{string}, is a formal. The first implementation of LINDA introduced in Gelernter’s seminal paper, was developed using C as the host language and used a syntax like this: \texttt{in(1, a:int, b:float)}. This would request a 3 element tuple, the first element being an integer 1, the second element an integer and the third a floating point number. Although not supported by C, this syntax was converted by the LINDA compiler to valid C code.

LINDA has three primitive operations which are described below. Other tuplespace based systems have similar, if not identical, primitives.

- **out(tuple)** Takes the tuple and adds it to the tuplespace.

- **in(template)** Returns a tuple matching the template. The matched tuple is removed from the tuplespace. If no tuples matching the template are currently in the tuplespace then the call blocks until a tuple is placed in the tuplespace that does match. If there is more than one matching tuple it is unspecified which tuple will be returned. If more than one process calls \texttt{in} with the same template then it is unspecified which process will get the matching tuple.

- **rd(template)** Identical to \texttt{in} but creates a copy of the tuple and leaves the original in the tuplespace.

Given that Linda specifies that a tuplespace must be accessible from any location it is clear that it is distributed across space, as two processes running on separate computers can communicate. Linda is also, in some senses, independent of time. A process will block on a \texttt{in} or \texttt{rd} until such time as an \texttt{out} is performed by another process. In addition a tuplespace is a persistent medium so a process can output a
tuple in the knowledge that however long it is before another process tries to access it, the tuple will remain available. Information is not assumed to be communicated instantly between processes. The information is ‘buffered’ by the tuplespace and can be picked up as and when needed by a process.

There are fundamentally two different types of Linda system: open systems and closed systems. The difference is related to what programs are allowed to access the Linda network. A closed system requires all programs to be compiled at the same time, possibly along with information about what processes will run where. This gives the compiler information that enables it to optimise the locations of tuples. It also has complete information about the types used.

An open implementation of Linda allows any program to connect to the Linda system, with no prior knowledge needed by the compiler or the Linda kernel. This obviously limits the type of optimisations that can be performed to those that are done dynamically. The benefits of using an open implementation are that the system is far more flexible in how it is compiled and run.

A tuplespace is described as a shared memory structure. This means that all processes, wherever they are in the network, must have the ability to access it as if it were local to them. The LINDA model does not specify whether the tuplespace is physically shared or not. The simplest method of implementation is to have all tuplespaces stored on a single ‘master’ server. Every other server on other nodes simply acts as a proxy, forwarding messages between the client process and the master server. As the number of clients and nodes grows this structure will not scale well. In many practical implementations the tuples of a single tuplespace are spread between machines, thereby partitioning the tuplespace. This partition is often implemented using distributed hash tables [34] but the exact method of storing tuples does not effect the changes to LINDA that are proposed in this thesis.

In a closed system the tuplespaces can be held inside the processes that make up the system, even though they will appear to be separate and accessible to all. In an open system there is not enough known at compile-time about the characteristics of the system so a separate process is required to hold the tuplespaces and direct the flow of values between processes. Regardless of whether the tuplespace is physically separate to the client processes or not it is logically contained within the LINDA kernel. In the case where tuplespaces are separate there may be one or more kernel processes, which communicate between themselves and the client processes. Each client process talks to a single kernel process as it knows nothing about where the tuples are stored, or other processes in the system. Kernels will have much more information, such as ‘names’ of tuplespaces and processes, and will be able to direct values to the right tuples or locate them based on a template.

### 2.1.3 Extensions to Linda

Since the original specification for LINDA was proposed, a variety of implementations have been produced, each supporting a differing set of extensions that enhance the
functionality of a basic LINDA system.

2.1.4 Multiple Tuple Spaces

The initial specification of LINDA had a single, global tuplespace available to all processes in the LINDA system. One of the most obvious extensions to this model is to allow multiple tuplespaces. As Gelernter [25] and Hupfer [31] discussed all that is needed is to add a new Tuplespace type (sometimes, as in ISETL-Linda [29], called bag) and a new CreateTuplespace primitive, and to modify the existing primitives to accept a Tuplespace parameter.

Figure 2.1 shows two differing models of multiple tuplespaces graphically. Initially, multiple tuplespaces were implemented as a hierarchy [30] but the trend has been towards multiple disjoint tuplespaces (as in Laura [28]). It is believed that this is because the flat model is more flexible; the added structure of the nested model is useful in some situations but counterproductive in others. In a system using nested tuplespaces it is possible to partition the tuples into two or more groups and allow processes to work within their own partition. Other processes can work in the tuplespace which gives them access to the tuples in all of the partitions. It is not intuitively clear what happens to a tuple which is placed into the higher level partitions. Is it visible in one, all or none of the partitions? Confusions such as this lead to a greater number of mistakes by the programmer, while the simplicity of the disjoint model makes it very easy to use.

Taking the nested tuplespace model further another possibility is the use of scopes [35]. Intuitively a scope is like an overlapping tuplespace that gives a view on some or all of a group of tuples. The ability to partition a tuplespace gives the ability to easily replicate the functionality of the collect and copy-collect primitives (which are discussed below). Using scopes to partition a tuplespace improves on the collect primitive by moving tuples between different scopes of the same tuplespace, this allows a solution to the multiple rd problem [26, 36] (which is described and solved by different means below). It still allows processes that have access to the whole tuplespace access to the tuples at all times.

The flat model of multiple tuplespaces is the simplest and most common method
for adding multiple tuplespaces and is the model that will be used in this thesis.

2.1.5 Non-Blocking Primitives

One of the features of the original Yale version of LINDA [3] was the inclusion of duplicates of the in and rd primitives, inp and rdp. Instead of blocking if no tuples matched the template, they return a special value indicating no match. This has the significant problem of breaking the time independence of a LINDA program. If, when inp and rdp are called, the tuplespace happens to contain no matching tuple then the no match value will be returned. The usual way of using these primitives will make the program exit; even if a matching value may appear at some point in the future. The question of when something happens in an asynchronous system is a tricky one, as there is no easy way to ensure all partitions of a tuplespace are observed at exactly the same point in time.

It feels unnatural to write code that could block forever, but it is usually possible to rewrite the code in a form that will work with the blocking primitives, or the bulk tuple operations described in section 2.1.6.

In [37], using logic similar to CSP [38], Jacob and Wood develop a semantics for the nonblocking primitives that maintains the time independent properties of LINDA. An intuition for the semantics that they develop is that the operations do block until a clique of processes, deadlocked on a clique of tuplespaces, is detected. If this clique is made up of all processes which have knowledge of all the tuplespace in the clique of tuplespaces then no progress can be made until one or more of the processes is reactivated by making an inp call return without matching. This will require a kernel to maintain information about which processes know about which tuplespaces and whether they are blocked on an inp or not. Most of this information will already be being collected for the purpose of garbage collection and so is relatively straightforward to implement.

2.1.6 Bulk Tuple Operations

The classical LINDA model suffers from a problem known as the multiple rd problem [26,36]. If a process wishes to perform a non-destructive read on a subset of tuples then in the classic model all the tuples must be destructively read, processed and then returned to the tuplespace. This is clumsy and error prone especially if other processes are adding and removing tuples while this read is taking place.

The multiple read problem is a result of the non-deterministic nature of LINDA. The semantics of LINDA do not specify which tuple you will receive if there is more than one tuple that matches the template being matched. They do not even specify that you will receive a different tuple if you perform more than one read on an unchanged tuplespace. Indeed a common implementation of the rd primitive is to return the first matching tuple in whatever data-structure is being used to store the tuples. If the tuplespace does not change then the same tuple will be returned.
each time. This makes it difficult to count the number of tuples matching a pattern effectively.

In [26, 36] a pair of new primitives were proposed collect and copy-collect. Given two tuplespaces \( t_1 \) and \( t_2 \), \( \text{collect}(t_1, t_2, \text{template}) \) moves all tuples that match \( \text{template} \) from \( t_1 \) to \( t_2 \). The function copy-collect is similar except that it copies, rather than moves, the tuples between the two tuplespaces. Both primitives return an integer representing the number of tuples that were placed into \( t_2 \).

```c
void monitor() {
    ....

    ts = create_tuplespace();

    int count = copy_collect(workspace, ts, template);

    if(count == max) {
        /* finished */
    } else {
        /* carry on working */
    }

    ....
}
```

Figure 2.2: Example Of Using A Bulk Tuple Primitive

Figure 2.2 shows an example of a program using the copy-collect primitive to monitor the progress of another process. This would only have been possible previously by modifying the tuplespace, using copy-collect means the monitoring is non-invasive.

A side benefit of using these new primitives is that the efficiency of a distributed implementation is increased. If the operations are performed using repeated in or rd calls then many packets will be sent across the network. With a bulk tuple operation these packets can be condensed into (ideally) a single packet. Rowstron’s 1996 paper [4] contains a quantitative demonstration of the speed advantage of transferring multiple tuples per packet between kernels.

Other hybrid coordination systems have solved the bulk tuple problem differently. The highly distributed TuCSoN system [39] has programmable tuplespace in which actions, specified in the ReSpecT language [40], can be associated with events that occur in the tuplespace. When triggered, these actions are performed in a single transaction so the state of the tuplespace remains consistent throughout its execution. The action can cause further events to occur in the tuplespace, and can modify the state at will. To duplicate the effect of the collect and copy-collect primitives an action can be associated with a certain type of tuple, which when ousted into the tuplespace, copies or moves all tuples matching a specification (included as part of the original tuple). The original tuple is discarded rather than added, and
if required a new tuple is created specifying how many tuples were moved.

Similarly Mobile Agent Reactive Systems [41] (MARS) adds reactive tuplespaces to a mobile agent system with LINDA-like coordination. The MARS architecture allows the use of Java based mobile agents over a wide heterogeneous network (such as the Internet.) Each mobile agent is given a reference to the local tuplespace when it arrives on a given server. The tuplespace is associated with a meta-level tuplespace which manages the reactions that are connected to actions performed on the tuplespace by the agents. There are four items of information that form the association between action and reaction. They are the reaction, tuple, operation type and the agent identity. For example the tuple \((\text{ReactiveObj}, \text{null}, \text{read}, \text{null})\) associates \text{ReactiveObj} with all \text{read} operations regardless of the template used or the agent that performed the operation. The multiple read problem can be solved in a very similar fashion with MARS as it was with TuCSoN and ReSpecT.

2.1.7 Garbage Collection

In an open implementation of LINDA, with multiple tuplespaces, a major problem is memory usage. If a system implements the bulk tuple operations then a common design pattern is to create a new tuplespace, copy some tuples to it, count how many there are and then lose the handle to the tuplespace. If no garbage collection is implemented then this tuplespace will remain in existence indefinitely despite no process ever being able to access it again. It is obvious that this situation will quickly cause the system to run out of memory and fail. Also, just as open distributed systems support transparent, uniform placement of objects they should also support transparent object management including deletion of unreferenced objects.

Local garbage collection is a well understood problem, and has been implemented in many languages but expanding it to a distributed system is a challenge. Tracing algorithms require costly termination mechanisms while reference counting [42] is scuppered by common message failures. Major goals of distributed systems are efficiency, scalability and fault-tolerance, but most methods for performing distributed garbage collection only achieve some of these goals.

In their 1995 survey paper [43] Plainefossé and Shapiro describe the various techniques one can use to perform garbage collection on a distributed system. The simplest method is to extend the classic reference counting garbage collection scheme. If a process creates a stub representing a remote object then it must send a message increasing the remote object’s reference count, and similarly when the stub is destroyed. While this method is adequate in a theoretical system, it does not behave correctly in a system in which faults can occur and messages arrive out of order.

In his PhD thesis [44] Menezes describes a system to perform garbage collection in the LINDA-like system, \textit{Ligia}. He also deals with the related problem of implementing I/O operations in LINDA. To ensure that garbage collection takes place correctly it is split into three distinct phases. The system must have the information required to perform garbage collection so a check-in and check-out system for processes is
required. The first phase is tuple scanning, which detects when a reference to a
tuplespace is transferred. Secondly information is collected for garbage collection
on each node of the system based on the local tuplespaces. Finally a global garbage
collector is occasionally run to build a combined version of the local garbage graphs
and determine any global tuplespaces that are garbage.

2.1.8 Types

Type matching is a core part of LINDA as it is how programs retrieve data from
tuplespaces. Type information must be transferred with tuples when they are placed
into a tuplespace but the LINDA model does not specify what form this must take,
how complex the type information can be or even what types can be used.

The fact that the type information used in LINDA systems is not defined is
a disadvantage when compared to other distributed systems and most, if not all,
current implementations only allow very basic types to be used in tuples. For most
systems this means only atomic types and other tuples are allowed to be elements of
a tuple. While restricting types to purely atomic types is a valid option it does not
fit well with the high level nature of LINDA as it forces programmers to manually
decompose their types in such a way that they will be accepted by the LINDA system.
In a closed LINDA system it would be possible to allow complex types to be used
as the necessary conversions could theoretically be generated at compile time. The
benefits of allowing arbitrary processes to connect in an open system are compelling.
In addition the problems of handling a closed system with multiple programming
languages, some of which may not be compiled, are numerous.

2.1.9 Conclusion

LINDA is a very abstract system with a very high level interface. This makes coor-
dination between distributed processes very straightforward, and unlike most other
systems it does not require all programs to be compiled together nor does it make
any requirements about the languages used by the client programs.

When compared to the other coordination systems, that will be described later,
LINDA is very flexible and easy to use, but it would benefit greatly from being able
to use complex types in tuples. Consequently, although the outcomes of the work
proposed in this thesis will remain general enough to be applied to any system,
LINDA will be the one used to develop an implementation of the work.

2.2 COM and DCOM

DCOM [10] is the distributed extension to Microsoft’s Component Object Model [45]
(COM) technology. DCOM is a vast technology covering all aspects of distributed
computing, including features such as load balancing, fault tolerance, security and
integration with other Internet protocols. For the sake of simplicity only aspects of
DCOM relating to types and programming a DCOM application will be discussed here.

In the early ’90s Microsoft developed a technology known as Object Linking and Embedding (OLE). This allowed programmers to package their programs so they could be easily reused in other applications. Microsoft soon realized that cross-language compatibility and standardisation would be vital for this technology to be a success, and hence COM was born. In 1996 Microsoft introduced a set of extensions called DCOM that allowed COM components to be accessed in a distributed fashion.

Despite it being a well specified technology take up has been slow on non-Microsoft platforms, although it is available on both Unix and Mac. As it is built into all recent versions of Microsoft Windows it is still the most prolific of any component-based architecture available today. Lastly, COM is a single vendor solution, which while not relevant in a technical comparison, may pose a problem for users of the system.

2.2.1 COM Architecture

DCOM uses a client/server architecture and provides three different types of server representing the desktop orientated ancestry of DCOM. A DCOM server can be either in-process, local or remote. An in-process server is when the server is in the same address space as the client process. A local server is in a different address space from the client application, but is still on the same physical machine. Finally, a remote server is on a different physical machine from the client process. A client application is not required to know what kind of server it is using, once a process has obtained a handle for an instance of a COM object then all servers appear the same.

2.2.2 Types In COM

COM provides a restricted set of types but it does allow all the usual base types such as short, long and float. COM also supports enumerated types, structures and arrays amongst other compound types. The greatest limitation of COM is the requirement that only automation types are allowed to be used if any COM interface is to be used by an application that is not programmed in C++. Automation types consist of short (16-bit signed integer), long (32-bit signed integer), float (32-bit floating point number), double (64-bit floating point number), BSTR (basic string), VARIANT_BOOL (short integer used to indicate true or false), DATE (64-bit floating point number representing the fractional number of days since December 30th 1899), IUnknown* (generic interface pointer), IDispatch* (automation interface pointer), VARIANT (a discriminated union of all other automation types) and SAFEARRAY (variable sized multi-dimensional arrays). An obvious omission from this list is a type which allows the programmer to create a compound type. This is not a mistake: COM does not allow the use of custom compound types as parameters or return values from functions in automation safe interfaces.
As stated earlier if an interface is allowed to be accessed by clients other than C++ programs then they must only use *automation types*. There are few of these types and they are all limited atomic types. This means that any problems with different representations can easily be overcome by statically coding all appropriate translations, avoiding any issues with different architectures or languages.

### 2.2.3 COM’s Interface Definition Language

COM uses an IDL to specify the external interfaces for COM-enabled objects. Due to Microsoft being the vendor behind COM its tool support is excellent. Although COM actually uses an IDL it is not usually necessary to manually create the IDL file as the development tools handle the creation automatically. Once an IDL file has been created it is compiled into a binary *type library*. The type library is then registered with the Windows system registry\(^2\). This type library is then shared between all processes which use types specified in the type library.

### 2.2.4 DCOM Conclusion

DCOM benefits from being a single vendor solution in that as the tool support for it is so good programmers rarely have to worry about writing their own IDL files. The biggest problem with DCOM is that the ‘solution’ to the problem of incompatible type representations is an IDL with only a very limited array of atomic types. This only avoids the problem by making things so restrictive for the programmer that the problem no longer occurs.

DCOM is being phased out in favour of Microsoft’s newer .Net Remoting technology, which is discussed in section 2.5.

### 2.3 CORBA

CORBA [9] is an distributed object system which has support for almost every combination of hardware and operating system in use today. It also has mappings to a wide range of programming languages including C, C++, Java, Python, Smalltalk and Eiffel. C++ is probably the most common language that is used with CORBA and a detailed description of fundamental CORBA concepts and how to use C++ with CORBA can be found in [46]. CORBA is often compared to COM, which was discussed in section 2.2. Two excellent comparisons of the two can be found in [47] and [48].

CORBA is a fully ratified specification of the OMG\(^3\), an open membership consortium that creates and maintains specifications for inter-operable enterprise applications. The OMG was formed in 1989 to accelerate the introduction of standardised

---

\(^2\)Non-Windows implementations of COM provide their own version of the Windows registry and the appropriate system calls

object-based applications and to help create a component-based software marketplace.

2.3.1 An Overview Of CORBA

The key features of CORBA can be broken down into the following segments.

- Operation invocation and dispatch facilities
- Object adaptors
- Inter-ORB protocol
- OMG Interface Definition Language
- Language Mappings

The first three items will be briefly described, and then the final two items will be discussed in more detail as they describe CORBA’s distributed type system.

![Figure 2.3: The CORBA architecture](image)

The general structure of a CORBA system can be seen in figure 2.3. The fundamental object in any CORBA system is the ORB which controls all communication between processes. The ORB also provides Object Adaptors to the client programs which are essentially proxy objects. The proxy objects allow a client program to access an object as if it were local, but instead of performing the computation the proxy object invokes a remote procedure call and gets another machine in the network to do the work.

CORBA is a client/server architecture, but the distinction between clients and servers can only be made during a single request (a single invocation of an operation on a CORBA object by a client) as the client for one request maybe the server for another.

An object reference is a handle to a CORBA object that is used by a client process to identify a remote object and invoke methods on it. Object references are opaque to clients, they cannot be created nor can they be decomposed to their constituent parts. A servant is the instance of an object to which an object reference refers. Servants are said to incarnate CORBA objects as they are the actual objects
that receive any method invocation that is performed on an object reference by a client.

### 2.3.2 OMG Interface Definition Language

The fourth element in the list of the key features of CORBA was the OMG Interface Definition Language (IDL). In order to invoke the methods of a remote object a client must know the interface it provides. An interface is wholly defined by the operations it provides, the types of data it can be passed and that it returns. IDL is a declarative language that gives the programmer the ability to specify the interface to an object that will be available through CORBA.

A simple example of the IDL is this phone book interface:

```idl
interface Phonebook {
    long getNumber(in string name);
};
```

The interface `Phonebook` contains a single operation, `getNumber`, which takes a `string` as an input parameter and returns a `long`. An application could use the `getNumber` operation on a `Phonebook` object to return the number given a person’s name. All parameters to functions in IDL are required to specify their `direction`, either `in`, `out` or `inout`, so the ORB knows how to handle them.

IDL supports inheritance, in fact all interfaces silently inherit from the `Object` interface. This special interface supplies the basic operations supported by all CORBA objects.

### 2.3.3 Types in CORBA

The IDL language is essentially a method of specifying the format for types defined by a client program. IDL allows compound types to be built up using the `interface`, `struct` and `union` structures (amongst others). These all define compounds, and in any language there are core, or base, types that all the compound types are defined in terms of. CORBA is no different and defines all the types that you would expect. All of the CORBA types sizes are defined by a lower bound, both in terms of the actual number of bits and data range that it must support. The reason for this is that some computer architectures do not support all the types in the sizes required. Some do not support an 8-bit character type for example, so a larger type must be used. Also, some languages cannot support the ranges required. Java, for example, does not have unsigned integers and therefore the Java language maps `unsigned long` and `long` to the Java type `int`.

All basic types in CORBA (except `octet`) may undergo a transformation during a transfer between heterogeneous hosts. These transformations are implementation dependent, but will usually include byte-swapping on integers being transferred between `big-endian` and `little-endian` machines. Character data may also be transformed between encodings, such as ASCII and EBCDIC. Floating point numbers are
implemented using the IEEE single- and double-precision standards. If for whatever reason an implementation cannot support this standard then it must document any deviation.

Recent additions to CORBA include wide character support which allows character encodings such as Unicode. The standard deliberately does not specify the encoding to be used by the wide character types. Instead it is left to the implementation to choose the closest encoding to the native standard and to specify how this encoding is to be converted between hosts.

2.3.4 Language Mappings

The IDL is just a declarative language: before it can be used it needs to be mapped onto the language that the client application is being written in. For each IDL construct, a language mapping specifies how the host programming language is used to implement it. For example the Java language mapping defines that IDL interfaces are mapped to Java’s public interfaces. Language mappings also define how applications use ORB services and how server applications implement servants.

Before a language can be used with CORBA it requires a language mapping to be created. The majority of CORBA implementations provide mappings for Java and C++. Other object oriented languages with similar features to Java and C++ are well supported, including Python\(^4\), Smalltalk and Eiffel. The OMG provide [49] a specification for language bindings to Ada, C, C++, COBOL, CORBA scripting language, Java, Lisp, PL1, Python and Smalltalk. Despite the wealth of official specifications of language bindings, programming paradigms other than Object Orientation are not so well supported. The popular procedural language Fortran-90 has had some attempts made to support it [50, 51]. Unfortunately both these papers create C/C++ interfaces to the Fortran code and then interface the C/C++ code to CORBA. This use of an intermediary layer of code imposes an unnecessary restriction on both the code that is allowed and the performance of the compiled system.

One non-imperative object oriented language that is officially supported by CORBA is Lisp [52]. Lisp is a very powerful and flexible language, and this flexibility allows CORBA to be mapped fairly cleanly. All the basic types of CORBA map onto the equivalent Lisp types, for example strings map onto the Lisp string type, floats and doubles map to the real type. The CORBA types wstring and wchar have no direct equivalent in Lisp and so are simulated with the type corba:wstring which is a sequence with elements of type corba:wchar.

Here is an example of how CORBA maps to Lisp, taken from the official CORBA language mapping for Lisp [52].

\(^4\)OmniOrb is freely available and provides both C++ and Python mappings - http://omniorb.sourceforge.net/
IDL

module example {
    interface face {
        long sample_method(in long arg);
        void voidmethod();
        void voidmethod2(out short arg);
    }
}

Example of usage in Lisp

; assume x is bound to a value of type example:face
(sample_method x 3)
> 24
(voidmethod x)
> ; no values returned
(voidmethod2 x)
> 905 ; this corresponds to the out parameter

Despite CORBA and its IDL being principally designed for imperative object orientated languages, it is still mapped to Lisp fairly easily. The biggest problems with the mapping are the fact that not all the built-in CORBA types have equivalents in Lisp and that the out and inout specifiers for parameters are not easily handled. In the example given above, an additional parameter is added to the start of each function (x) to take an object of the type example:face. The sample_method takes a single parameter in the IDL file, so when called from Lisp it takes two parameters. The third function, voidmethod2 shows the problem with handling out parameters. Although in the IDL file the function does not directly return a value, and has a single out parameter, the signature is changed in the Lisp mapping. When called from Lisp it is given a single parameter (of type face) and returns a value. Situations like this become even more complicated if a function has more than one out parameter.

2.3.5 Conclusion

CORBA is a widely supported language, with excellent cross-platform support and, providing the programmer accept its restrictions, excellent cross-language support as well. The restrictions it imposes are CORBA’s greatest flaw. As has been shown CORBA requires types to be defined in a third party language which is then mapped onto the host language. This mapping is required to take place at compile time, so there is no possibility for a Linda-like open system, in which various programs not necessarily developed together, can communicate.

For an open system to be implemented using CORBA it could not use the IDL as that defines interfaces at compile time rather than at runtime. Although CORBA has the ability to construct requests dynamically at runtime this is not integrated
with the language used by the programmer and requires detailed knowledge of the CORBA API.

### 2.4 Comparison of Linda, CORBA and DCOM

Both CORBA and DCOM share a common goal. They enable programs to communicate simply, and easily using a variety of languages and on a variety of hardware platforms. They differ from LINDA in that they are not designed as an open system. CORBA requires types to be created from an IDL which is shared between all processes. Although not explicit, DCOM’s use of a type library is equivalent to CORBA’s IDL.

DCOM only has partial support for languages other than C. By forcing the use of only automation types when communicating with other languages it removes any benefit the type library might provide when using complex types. CORBA’s support for different languages is much better than DCOM, but as was seen with the example for Lisp it sometimes leaves a lot to be desired.

The requirement that in an open LINDA system connected processes know nothing of each other, and that they may join and leave at any time, make it difficult for either CORBA or DCOM to be adapted for use as a LINDA system. The method of sharing type specification at compile time is also incompatible, and so the only alternative, to share type specifications at runtime, must be used.

### 2.5 Microsoft .Net

Microsoft’s .Net Framework [53] is a platform agnostic virtual machine, class library and type specification system which provides developers with a rich development environment and a stable, secure and predictable platform to run their code on. The .Net Framework is built on top of the Common Intermediate Language, formerly known as MSIL (CIL) which is an object-orientated, stack based virtual machine. When run CIL code is verified for safety which provides better security and reliability than binaries native to the hardware platform.

On top of the CIL a number of different languages can be run. Unlike Java, .Net’s virtual machine was designed from the ground-up to support many languages. Languages such as C#, VB.Net, C++, Python\(^5\) and others are compiled to the CIL bytecode and can be run on the same virtual machine. As part of its language neutral design the CIL allows languages to import modules written in other languages and to use them as if there was no language difference.

Despite being an European Computer Manufacturers Association (ECMA) standard [54] CIL is strongly tied to Microsoft Windows. There are competing implementations for both Linux and Mac OS X but they are not as featureful as the primary Microsoft implementation, and for copyright and patent reasons they do

---

\(^5\)Through its IronPython implementation
not implement the entire .Net class library.

Although any language running on the .Net framework can import code written in any other language and use it there are inevitably some caveats that limit the programmer. IronPython has had to implement special semantics for the modification of an object’s member variables and on what can be thrown as an exception. Python does not support value types, only references to values. In Python, when you modify an object’s member variable the changes are is applied to the object but in C# retrieving a member variable, which is of a value type, creates a copy. In this case changes would be made to the copy and not reflected in the original object. As Python and C# also have different exception hierarchies, the implementation tries to automatically convert between them, so programmer in each language see the exceptions they were expecting. For example, if a Python programmers calls a C# function which raises a EndOfStreamException the implementation will automatically convert that object to an EOFError. The visa-versa is also true.

The Common Language Specification (CLS) is a set of rules and restrictions that if followed ensures that code written in a CLS compliant language can be used from any other CLS compliant language. Most languages include features which are not CLS compliant so compliance is dependent on the code as well as the language implementation [55]. Examples of the CLS rules that apply to compilers are that they specify a method for referencing identifiers that coincide with keywords in the language they implement. Also that they must support the case where a single type implements two interfaces which both require a function with the same name and type specification to be implemented. The programmer must take care not to break rules such as not using boxed value types, or their code will not be CLS compliant.

As part of the .Net framework a series of technologies are included under the heading of Windows Communication Foundation [56] (WCF). WCF provides a wrapper around SOAP [57] and other standard web-service technologies to give them a unified and consistent appearance as part of the .Net standard library. As well as the standard untyped message stream primitives, WCF provides a typed programming model (the service model) to ease development of distributed applications. WCF also implements high level concepts such as distributed transactions and managed object lifetimes.

In this section only managed interfaces will be considered as unmanaged interfaces are just an implementation of the SOAP standard. Services are specified using a service contract which consists of an interface or class with a special attribute set to indicate that it is a webservice. A service contract indicates what methods may be called on a particular interface. The data that is transferred is specified by a data contract. As with a service contract, a data contract is specified by applying a special attribute to the members of a class or interface. The standard .Net serialisation methods are then used to encapsulate a value of this type into a SOAP call.

By specifying the data to be transferred using attributes, WCF avoids the need to share full type information, but instead relies on the sharing of data contracts
and appropriate implementations being used at both ends of the communication. Data contracts are simply reduced type specifications. If the implementations do not produce exactly matching XML then a SOAP fault is raised.

The WCF does not provide much beyond a more modern implementation of typed messaging than DCOM. The ability to interact with arbitrary SOAP services is interesting for programmers but no novel methods of handling typed data are introduced. The ability of languages to import values of types from any other languages running on top of the CIL is a powerful tool but the range of types that can be used in this way is limited (although less so than COM’s automation types). Complicated type that use techniques such as multiple inheritance cannot be used cross-language and even simple things like unsigned types are not supported.

2.6 Mobile Agents

Setting aside Linda for one moment, many distributed systems are static at both a physical and a logical level. While advances such as wireless networking (802.11b, Bluetooth etc) allow a certain degree of physical mobility (i.e. the computer can be moved to another location while still communicating with the network) the logical view of the network remains the same. A mobile agent [58] is a specific type of software agent. It is a process that may choose to migrate from node to node in a network of computers and thus alter the logical make up of the network. An agent performs this move by saving its current state and transferring code and data to another machine where the execution is resumed.

Systems that allow the execution state of a process to be transferred along with the code are said to support strong mobility. If the execution state is discarded during migration then the system supports only weak mobility. While strong mobility has many advantages, including the fact that migration appears to be a single operation to the programmer, many systems only provide weak mobility. In this case the notion of what constitutes a mobile agent’s execution state is difficult to determine so in most instances it is left up to the programmer to specify what information should be transferred and how it should happen. This is due to a large proportion of mobile agent systems relying on the JVM and the features it provides for dynamic loading and running of classes.

Java has excellent support for at least some of the features required by mobile agent systems. Unfortunately, as stated in [58], this has the side effect of directing most mobile agent systems to use the same features and so many aspects of mobile agents that are not so Java-centric have been ignored.

2.6.1 CLAIM

One well known example of a system which uses mobile code is Computational Language for Autonomous, Intelligent and Mobile agents [59] (CLAIM). CLAIM is a language designed specifically to support both stationary and mobile agents.
Agents created using CLAIM are able to communicate with other agents, reason and are mobile. Taking inspiration from the Ambient Calculus [60,61] the notion of mobility is facilitated through a hierarchical series of scopes which contain executing processes. A scope can be subdivided into several other fully-fledged scopes. A scope can also be disbanded which causes everything contained within it to become part of the parent scope. A scope can also move to become a sub-scope of another scope, and it is this that allows the agents to move from machine to machine. CLAIM is implemented using Java to provide the fundamental technical support for mobile code.

The CLAIM language is designed to be a complete agent framework and consequently contains much more than simply mobile code primitives. As well as featuring a complete security model, CLAIM is designed so that the agents use complex components such as knowledge and goals to allow forwards and backwards reasoning and goal driven behaviour.

CLAIM is a fundamentally different system from LINDA in that LINDA treats client processes not as autonomous agents but instead as normal programs that happen to want to communicate with other, similar, processes in a simple and easy way. A CLAIM agent is not a simple process, it has an enforced structure that is imposed upon it by the CLAIM language. If we ignore the difference between mobile agents and simple coordinating processes then we can see a difference in their use of mobile code. While a mobile agent is allowed to move itself completely to another node in the network it can only communicate using a limited set of messages. The language itself makes no mention of types in messages implying that type handling is left to the underlying JVM. While CLAIM is interesting to look at from a mobile code perspective its reliance on Java’s JVM means its implementation suffers from all the problems associated with a single language implementation.

2.6.2 LIME

Although CLAIM implements a new language for writing mobile agents, other systems have modified existing distributed computing platforms to allow mobile agents. Linda in a Mobile Environment [62–64] (LIME) borrows and adapts the LINDA model of coordination to allow both physical mobility and logical mobility. The LIME system is a free and open-source\(^6\) Java-based coordination framework. LIME is designed such that it retains the basic philosophy and goals of ‘pure’ LINDA while adapting them to mobility. Programs view the world as a sea of tuples accessible by contents. Movement, be it physical or logical, causes an implicit change in the tuplespace accessible to the process.

In the classical view of LINDA, tuplespaces are persistent and globally accessible. Maintaining these properties in the presence of physical mobility is difficult. LIME is specifically designed such that is does not count the changes to a network caused by physical mobility as a fault, it is gracefully handled and appears to other users as

\(^6\)See [http://lime.sourceforge.net](http://lime.sourceforge.net)
though the node which moved never disconnected from the network. The motivation
for the changes made to LINDA by LIME is that in the presence of high amounts
of physical mobility (including, for example, mobile phones as part of the network) and
logical mobility the idea of a persistent, globally visible tuplespace is unreasonable.
MobiSpace [65] takes a similar view of the problems caused by physical mobility,
but restricts its design to focus on implementing for Java-enabled mobile phone-like
machines.

LIME’s model is that a mobile agent is a process that can travel between mobile
hosts. The mobile agents are the active components of the system, while the mobile
hosts are the roaming containers for the agents and provide connectivity for the
agents. The key notion that was added to LINDA as part of LIME is the transiently
shared tuplespace. Tuplespaces are permanently bound to mobile agents and mobile
hosts. Transient sharing enables dynamic reconfiguration of their contents according
to changes in the system topology.

Each process is permanently attached to an Interface Tuplespace (ITS) which is
transferred with the agent when movement occurs. Each ITS contains information
that the agent is willing to share with other processes. Each process accesses its
ITS through the usual LINDA primitives. Rather than only accessing the tuples
contained within the agent’s ITS, the primitives also access tuplespaces belonging
to other co-located agents. The tuplespace is shared by construction and transient
because the contents of the tuplespace (as it appears to the agent) changes with the
migration of agents.

As well as having the shared ITS each process may also have a number of named
private tuplespaces which are not subject to the same level of sharing as the ITS. If
an agent A has the tuplespaces Q, R and S while an agent B has the tuplespaces
S, T and U then only the tuplespace named S will become transiently shared be-
tween A and B. This is a significant alteration to LINDA as tuplespaces are no
longer anonymous and shared by passing references between processes. In LIME
tuplespaces are literally shared by name.

The transient nature of tuplespace sharing in LIME changes some fundamental
aspects of LINDA. In LINDA the individual processes are not aware of their location
or the location of the other processes. The LINDA network that binds them together
guarantees that the data placed into tuplespaces will be available to all other pro-
cesses that have a reference to the tuplespace until the item is removed. In LIME
if a process A wishes to send a message to a process with which it is co-located,
B, it can out a tuple as normal. In LINDA this tuple will still be available for B
to retrieve even after A has terminated. With LIME the tuple would be discarded
when A terminates as it is stored in A’s ITS thus when B attempts to retrieve the
tuple it will block[7]. To overcome this issue LIME introduces the notion of location
to the LINDA primitives. As well as taking the tuplespace handle and tuple as pa-
rameters the out primitive takes a reference to another mobile agent. It is the ITS

[7]Assuming the network only contains A and B.
of this agent which the tuple is placed into. This is another fundamental change to
LINDA’s semantics as processes are no longer anonymous.

Although LIME is based on LINDA and tries to maintain many of the advantages
of programming with LINDA the changes made to incorporate both kinds of mobility
are many, and complicate the programming interface. The development of LIME
was motivated by a desire to combine the simplicity of LINDA’s programming model
with the power of mobile agent technology. The notion of types is not discussed in
the work, nor is the notion of mobile code as a function. Indeed each agent is a self
contained unit whose only contact with other agents is through a tuplespace, as is
the case with the unmodified LINDA.

While the LIME papers makes no specific requirements for the implementation
language it does not deal with the issues that a multi-language system would intro-
duce. The lack of details on the type handling and the obvious, if unstated, reliance
on Java to avoid many of the issues that have already been mentioned mean that
LIME is clearly not merely the addition of the ability to move code between hosts.

2.6.3 Comparison To LINDA

Perhaps the most notable difference between vanilla LINDA and CLAIM and LIME
is that LINDA processes have no ability to control themselves (aside from causing
their own termination). Both CLAIM and LIME can be considered as giving a
process the ability to control itself as though it were a value. This level of control
is in addition to the standard primitives that a process uses to communicate with
other processes in the system.

As was stated in the introductory discussion of mobile agents, many systems
use Java as their sole implementation language. This is contrary to the spirit of
LINDA and while the JVM has many advantages for implementing mobile code it is
not suitable for a LINDA-like system. In those systems that do not rely on Java in
such an explicit fashion, the majority of the rest are new languages that have been
designed from the ground up for use in creating mobile agents. Again this is against
the spirit of LINDA as LINDA is designed to be added to any programming language,
to allow it to communicate with other languages in the LINDA network.

Adding this sort of additional control from a LINDA process is beyond the scope
of this thesis and as such will not be tackled. The work in this thesis focuses on
adding the ability to send and receive new types of values through the standard
LINDA primitives of in, rd and out. The closest that traditional systems have come
to mobile agent support is the eval primitive. As part of placing a tuple into a
tuplespace this spawns a new concurrent process somewhere on the LINDA network.
The elements of the tuple are replaced the results of the evald processes when they
terminate.

Although eval allows a process to request a new process to be spawned at some
location on the network, and to control what code the process will execute, it not
the case that a partially executed process is moved. The new process is created and
executed exclusively on a single node. Although the location transparent nature of
the LINDA network means that a process could be moved without any other processes
being affected, this kind of moving, if it occurs at all, is purely part of an automated
load balancing scheme.

The eval primitive does require code to be transferred from the process that
called the primitive to the machine that the process will be executed on. This clearly
satisfies the requirements for being ‘mobile code’ and implementing eval forces the
designer of the system to impose many of the same limits on the programmers. Like
many of the system mentioned previously, the client programs may be forced to run
on top of a virtual machine, such as the JVM, or the system may be forced to be
run upon a homogeneous network where the binary machine code can be transferred
and run.

2.6.4 Agent Communications

Agents communicate using an Agent Communication Language (ACL), which de-
fines how the agents will communicate. Languages such as the FIP A Agent Com-
munication Language [66] and KQML [67] are divided into three parts, knowledge
interchange format, a language for defining sharable ontologies and a high level in-
teraction language. The knowledge interchange format defines the syntax of the
communication between agents. The high level interaction language describes the
pragmatics of the communication. The language for defining a sharable ontology is
the most interesting aspect for the work in this thesis. An ontology is an agreed
upon vocabulary and semantics for describing a given subject domain. Agent com-
munication is typically done at a higher level than the bits and bytes level of this
thesis, but it is interesting to note the similarities between sharing ontologies and
converting values between different type representations.

Ontolingua [68] is an approach for defining ontologies in such a way that they
can be converted between different formats while being maintained in a machine
readable format. With Ontolingua the user selects a predefined base ontology and
extends it into the specific domain that their agent system will work. Messages are
constructed using this ontology and passed between agents. As agents understand
how to create messages, and the semantics behind them, messages do not need to
be in an exactly set format as they are parsed and understood as they are received.

2.6.5 Conclusion

In this section it has been shown how mobile code and LINDA have been used together
in the past. Although these systems grew out of LINDA they do not stay true to
LINDA’s spirit and cannot be considered to be true LINDA implementations. The
problem of moving code in these systems is either avoided (by using Java) or ignored.
Avoiding the problem is far from ideal and is not an idea that will be entertained in
this thesis.
The `eval` primitive is an interesting because an executing function is conceptually placed inside a tuple. The function cannot be matched against as this is purely an intuition as to how the primitive works. This thesis will consider how a function can be placed inside a tuple and can be treated the same as any other value. This is something that is not possible in the LINDA-like system studied above.

2.6.6 XML

XML [69–71] is a general purpose mark-up language, recommended by the The World Wide Web Consortium (W3C). XML is designed such that languages, or ‘dialects’, are easy to create and process. It is also mostly human readable, making XML data easy to work with. XML provides a textual means to describe and apply a tree-based structure to information. All information is given as text, interspersed with tags, which separate the information into a hierarchy of character data, elements which contain a mix of character data and other elements, and attributes of those elements.

Many documents are still text related, and most of these documents have some kind of structure to them (i.e. they are not free-form text). XML provides a means of imposing a self-describing structure on a document. In a network situation the data returned after a request must be able to describe itself or the client will have to make further requests to try to decode the message it has been sent.

XML does not specify the names, allowed hierarchy, or meanings of elements. They are instead left as definable through the use of a customisable schema. By doing so XML provides the syntactic foundation for the creation of new, XML-based, mark-up languages. The general syntax of the languages is rigid as documents must follow the basic rules of XML so that all XML aware software can at least parse and understand the relative arrangement of information within them. Schemas simply add an additional set of constraints on top XML’s syntax rules.

An XML document has both a physical and a logical structure. The logical structure means a document can be divided into named units and sub-units, called elements. The physical structure allows components of the document to be stored separately; these are called entities and can be stored in separate data files - this enables non-XML data such as images or sounds to be included.

```
<flight>
  <flight-number id="BA001" />
  <destination>New York</destination>
  <origin>Heathrow</origin>
</flight>
```

XML is actually a meta-language as the meaning of the tags is described by another language. There is no pre-defined list of elements. A Document Type Definition (DTD) defines the elements that are allowed in an XML document, and then using a validating parser the document can be compared to its DTD.
DTDs provide only limited support for describing the ‘shape’ of an XML document. DTDs were introduced as part of XML 1.0, but developers quickly outgrew the capabilities that DTDs provide for validation and so in XML 1.1 a new form of validation, using XML Schemas, was introduced.

Types In XML

XML itself does not support data types. XML has no notion of how characters should be interpreted. The value “3.14159” is just a seven character string. Additional processing, over and above parsing by an XML parser is required to turn it into a floating point number. By not specifying how the character string “256” should be interpreted XML avoids any problems with byte-ordering or type sizes.

An XML schema [72] is a document which describes what constitutes a well-formed XML file for that particular language. Confusingly, XML Schema [73] is one such language and is the first schema language to achieve Recommendation status by the W3C. XML Schema is designed so that a document can be validated against a language described by an XML Schema file and the validation will result in a collection of information which adheres to specific datatypes.

XML Schema defines a collection of basic datatypes, including 8, 16, 32 and 64 bit signed and unsigned integers; single and double precision floating point numbers; decimals and booleans. It also has a Unicode string type, as well as a number of other types less suited for direct conversion for use in a programming language. The complete set of types can be found in [73]. These are called simple types. The language also allows the description of complex types, which contain more than one value. Complex types are either just a renaming of a simple type with additional attributes (a complex type with simple content), a C-like struct (a complex type with a complex content) or a discriminating union type.

The code below creates a new type, `digit`, which is an integer restricted to between 0 and 9 inclusive.

```xml
<xs:simpleType name="digit">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0" />
    <xs:maxInclusive value="9" />
  </xs:restriction>
</xs:simpleType>
```

One key flaw in XML Schema is the lack of support for types such as typed references and arrays. Although arrays can be defined by creating an ‘element’ type which is repeated inside a complex type it should be possible to define an array without creating an intermediate type. Although XML has means by which you can refer to other elements in the same document, much like a pointer, XML Schema’s type system does not include the ability to specify one of these references as a type.
Integrating XML And Programming Languages

If XML were to be a viable solution to the problem addressed by this thesis then it must integrate well with fully-fledged programming languages.

The functional programming language CDue [74] is derived from XDuce [75], and is a functional language designed specifically to work with XML data. The datatypes in both CDue and XDuce are directly related to the datatypes in an XML document. In addition they also introduce regular expression types to match XML elements. While CDue and XDuce show that the XML type system is fully featured and capable of being used as a type system for a fully fledged language. These languages are designed specifically to have XML as their type system and therefore this does not say anything about how adaptable the XML type system is at mapping other type systems onto it.

While CDue is designed specifically to work with XML, Wallace and Runciman have modified [76] Haskell to work with XML. They describe two methods for doing this, either by creating datatypes for Haskell that are independent of any DTD or by deriving the definitions of datatypes based on a DTD. If a well formed XML document validates under a DTD then by extending the Haskell type system using the DTD the process of validation can be extended to validating applications that process the documents as well. The second of the two options is closest to solving the problems addressed by this thesis as it creates types based on a DTD. This is no better than a solution such as CORBA as this requires a third party type system to be mapped on to the host language. In this particular case there is the problem of Haskell requiring types to be named in a certain fashion, something that is not required by XML. This means the XML→Haskell translation process manipulates the names of the XML types to make them fit the Haskell scheme.

In addition to functional languages the XML type system has been applied to logic programming [77]. The XML elements are translated to terms. The DTDs or XML Schema are translated into regular types and a type inference system that has been implemented in Prolog automatically performs the validation. While not of any great interest with respect to the goals of this dissertation this paper does show how the XML types system can be mapped onto a more esoteric language.

LINDA and XML

The 2002 survey paper [78] by Ciancarini, Tolkosdorf and Zambonelli discusses taking the XML document-centric model and applying it to LINDA. The principle the paper uses is that of active rather than passive documents. Although an XML file is simply a text document, through extensions to XML it can have procedural behaviour associated with it. Extensible Stylesheet Language Transformations [79] (XSLT) and Extensible Stylesheet Language Formatting Objects [79] (XSL-FO) are two examples of languages that are attached to a document and provide functions that can be applied to the document. In this respect the document becomes self-modifying, or active.
XML documents are traditionally static and this more usual form can also be applied to LINDA. The concept of an ‘XML Space’ is the general notion for a replacement of the tuplespace by an XML based system. Most systems that use the XML Space concept create an XML document for each XML Space and this is then manipulated by the processes. An XML based tuplespace system has the benefit of being able to store the tuples (or in some cases arbitrary pieces of XML data) in a well defined, universal format. The problem of type matching is also solved automatically as the XML data can be validated by an XML schema. If the data can be validated then there is a type match.

XML-Spaces [80] is an implementation of the XML Spaces concept, based on the TSpaces [81] Java implementation of LINDA. XML-Spaces does not make the tuple an XML document, but each element in a tuple is an XML document with the tuple being an separate object type. A single XML document can be added to the XML-Space. The design XML-Spaces is extremely focused on flexibility and does not specify how type matching on XML documents should be achieved. The implementation provides 11 different possibilities from exact matching, thorough subset matching, DTDs, XPath to XQL. There is no mention of the conversion of documents if the format used to request the document does not exactly match how it is stored.

**Querying An XML Document**

XML has been proposed as a possible data interchange format for use with databases. In order for this to work it is necessary to create a query language for querying XML documents [82,83]. A variety of languages were proposed by academics but the W3C has chosen XQuery [84] to be the official query language for XML. Merging SQL with XML produces a language which can match based on the shape and content of an XML element. This is analogous to matching based on the type of the element. These languages all require specially crafted code to be created by the programmer, to determine if the types match.

**Type Matching In XML**

Type matching with XML can be viewed as the process of validating some data against a schema. If the given piece of data is validated then it is of the type specified by the schema. This method of type matching relies on a well defined schema being present and used by all programs. There is currently no method of determining isomorphisms between XML schemas. This results in XML being simply a third party type system that has to be defined prior to any coding and then used in all communicating processes.

**Schema Matching**

Schema matching [85] is the process whereby two schemas are run through an algorithm which produces a series of mappings between elements in the two schemas.
There are three elements of a schema that can be considered when matching. They are the element names, the type of data that can be contained within the element and constraints on that data. Cupid [86] is such a matching algorithm which uses linguistic techniques to match the tag names of elements, as well as looking at the structure of elements. It is biased towards the leaf elements of schemas as this is where much of the semantic information of a schema is captured. A detailed description of the matching process used in Cupid is beyond the scope of this review, but it results in a match coefficient between 0 and 1 for each pair of elements. A mapping is then derived from these coefficients.

Schema matching is interesting because it enables the matching and conversion of documents that are specified using heterogeneous schema. In many respects, this is the problem that is that is being solved in this thesis. Schema are types while documents are values of those types and the conversion between different schema is the conversion of values between different representations of the same types. Many solutions to the schema matching problem use complicated heuristics based on the text of the element’s name. Cupid, for example, knows that UnitsOfMeasure may also be written as UoM. This is error prone and mismatches may be tolerated by programs that are designed to use XML documents directly, but may cause programs using types, that are directly mapped onto memory, to crash.

Conclusion

As a mark up language XML is extremely good. As a way of specifying a type system it leaves something to be desired. An XML Schema document is complicated to create by hand, and is a very verbose way of specifying types. Languages created solely for the purpose of specifying types are typically much more concise. Creating an XML Schema based on types already defined in a different language is laborious, and XML Schema’s type system does not have the commonly used pointer types.

Although XML Schema could be used as the method for describing types the problem still arises that different XML Schema would need to be matched, and values which validate against one, converted to validate against the other. Due the verbose nature of XML Schema, and lack of pointer types XML Schema will not be used for the type system in this project.

2.6.7 Web Services

With the advent of so-called Web 2.0 the use of standard web servers to form the basis of a distributed system has become more common. The area of web services is a mish-mash of competing technologies and methodologies. This section will focus on the two illustrative web service technologies, SOAP and JavaScript Object Notation (JSON)-RPC.

The ancestry of web services is in RPC, which was the name given to the earliest methods of calling functions on remote machines. Most modern distributed systems are based on RPC derived technologies. An RPC protocol consists of a method for
serialising values, transferring them along with a method name to a remote machine and then receiving and deserialising the result.

**SOAP**

SOAP\(^8\) [57] forms the foundation layer for a web services stack. It is a protocol for exchanging XML messages over a network, typically using HTTP. SOAP is the successor to XML-RPC, which used XML to serialise RPC requests and return values. Although SOAP was developed before the XML Schema language (discussed in section 2.6.6) became a full standard, the new type system was retro-fitted to SOAP so it no longer used its own type system.

As SOAP is used purely to transfer messages, and does not define its own type system, all the criticisms of XML apply to SOAP.

**JSON-RPC**

JSON-RPC [87] is an implementation of RPC encoded using JSON [88]. JSON, as the name suggests is a lightweight method for serialising values. It is text-based and human-readable with a limited set of types. It is based on Javascript’s [89] object literal notation, which means that in Javascript a JSON value just needs to be evaluated to convert it from the string representation into a Javascript object. The simplistic nature of the format has helped in encouraging adoption of JSON, and code exists to decode values in a wide variety of other languages.

The types supported by JSON are: number (which can include integer, real or floating point values), string (Unicode only), boolean, array (an ordered sequence of values), object (a collection of key/value pairs) and null. What is immediately notable is it does not distinguish between integers or reals and only allows Unicode strings. It does not specify the size of integers or reals and so does not handle the case where a programming language or hardware platform cannot represent a value in the JSON value (i.e. because it is too large).

JSON does not have a *type* type which makes it impossible to transmit type information. The type of the data is implicit in the value. This means that an integer value can only ever be of an integer type and not (for example) a integer, float sum type. The type of any JSON value is simply an infinite sum type contain every possible combination of the JSON types.

**Conclusion**

Web services provide a simple base for building more complex distributed systems. When combined with things such as Multicast DNS [90] or Zeroconf [91] processes can discover the names of servers which provide the service they require.

---

\(^8\)SOAP was originally an acronym that stood for both ‘Simple Object Access Protocol’ as well as ‘Service Oriented Architecture Protocol’ but in version 1.2 of the specification the acronym was dropped and it became simply SOAP.
Although such systems are type safe this is due to client processes being required to format their data in the exact format required by the server.

2.7 Logic Coordination

In this final part of the literature review coordination models that involve logic based systems will be discussed. Logic programming, in its purest form, involves the programmer declaring a set of truths about the world they are modeling and then asking questions to a theorem prover or model generator which uses those truths to provide the answers. Like other forms of programming, logic programming can benefit from distributed computing and coordination. In this section a brief overview is given of several distributed logic programming systems.

2.7.1 Shared Prolog

Shared Prolog [92] extends the standard Prolog primitives `assert` and `retract` to mean send and receive data from the shared workspace. Processes add and remove atoms from the workspace, the ‘blackboard’, as they work towards solving their goal. There is no time constraints on synchronisation between processes, process S can send an atom to the blackboard and run to completion before process R reads it.

A process in the Shared Prolog architecture is known as a theory, which is a standard Prolog program augmented with a number patterns which specify when the theory will be activated and what will happen post-activation. The first activation predicate is the In\_guard which is a set of atoms which are read and removed from the blackboard. If all atoms (under unification) are contained in the blackboard then they are read and removed. After the In\_guard has finished modifying the blackboard the computed substitution is passed to the Read\_guard which is a set of built in predicates and positive and negative atoms. If the In\_guard and Read\_guard the theory is said to be committed to activation and if the main body of the theory fails the modifications made to the blackboard are not undone.

While Shared Prolog’s blackboard metaphor is similar to LINDA’s tuplespace, the process of developing applications is very different. LINDA programs can run separately to the LINDA network, communicating only when they want to. Shared Prolog programs must follow a strict pattern of specifying the state of the blackboard before they can be executed, the program to be executed and then their return value(s).

2.7.2 SICStus Prolog Linda

SICStus Prolog Linda [93,94] is a more traditional LINDA system than Shared Prolog and it maintains the classic trio of in, rd and out primitives. It also has the traditional client/kernel design with tuplespace partitions being stored on the kernel while client programs interact with the tuplespaces. The main difference that is
introduced by implementing Linda in Prolog is that the tuplespace becomes deductive. As Prolog rules as well as facts are able to be placed into the tuplespace, and the rules operate on the stored facts in the tuplespace in the normal Prolog way, \texttt{in} and \texttt{rd} operations can be satisfied by facts that are inferred by the stored rules.

As Prolog is untyped SICStus Prolog Linda’s tuples are also untyped. Tuples are partitioned amongst the servers that make up the Linda network by specifying a many-to-one mapping of tuples onto tuplespace partitions. This is similar to the idea of multiple tuplespaces, but this implementation limits the sorts of tuples that are placed into each partition. Most implementations of Linda have \texttt{in} and \texttt{rd} taking the tuple template as a parameter and returning the matched tuple. Prolog Linda uses the much cleaner method of specifying formal template parameters as unbound variables. After matching the variables become bound.

2.7.3 tuProlog

tuProlog [95, 96] is an open source Prolog engine designed for use in internet based distributed systems. It is implemented in Java to allow it to be integrated easily with other components. Remote interaction with a tuProlog system is possible through TCP/IP, Remote Method Invocation (RMI) and CORBA. The most interesting aspect of tuProlog is its ability to integrate with tuplespace based coordination models, including the ReSpecT language which was discussed in section 2.1.6. Both Java and tuProlog can interact with the TuCSoN tuplespace system [39] which is based around ReSpecT and provides seamless communication between both languages.

As tuProlog is developed in Java and one of its key aims is to take advantage of the Java standard class library, and for Java programs to be able to include the tuProlog engine as a class it is not surprising that tuProlog uses Java’s type system for all its interactions with other processes.

2.7.4 Conclusion

This section has given a brief overview of the systems which integrate tuplespace-based coordination models with logic programming languages. Logic programming languages are implemented in a way which is so different to the way that computer processors actually work that it is difficult to make a meaningful comparison without giving a much greater portion of this thesis to that discussion. The notion of type in logic language is often dependent on the implementation of the language, such as tuProlog’s use of Java’s type system.

The systems mentioned in this section avoid the problems of type matching by using untyped tuples. The systems are implemented using a single language which is far enough removed from the underlying hardware so that it does not introduce any heterogeneity into the system. Untyped tuples simplify the coordination model but at the expense of changing the way users interact with the system. Untyped tuples are such a large alteration to the tuplespace coordination model that they will not be considered further.
2.7.5 Types

Data types are a part of all programming languages [97,98], whether they be explicit or implicit in the definition of that language. A data type is a constraint on how data in a programming language may be interpreted. If a value, $x$, is given the data type `int` then that value can only be interpreted as an integer, and not as, for example, a string even if their the internal representation of an integer is also a valid representation for a string.

Programming languages typically have their own unique type system with its own quirks and peculiarities. C’s type system is heavily dependent on the programmer specifying what types they are using and on the types used by the underlying hardware. This contrasts with ML’s type system [99] which uses the Hindley-Milner to infer the types of variables and functions so the programmer can create a program without specifying the types they are using, but the program remains type safe.

The strictness of the type constraints imposed by the language and how they are evaluated affect the typing of the language. In some cases languages evaluate a different algorithm based on the type used. This is known as type polymorphism [100, 101] and allows the programmer to abstract algorithms away from the type of the data being processed.

The practical benefit of typing is that programs can be checked for correctness before they are run. Depending on the type system used this can detect a few of the most obvious errors (C, C++) or be a proof that the program will run without error\(^9\).

Some languages are not type checked until the programs are run. These dynamically typed languages typically provide more flexibility and a faster development cycle than languages where data types must be declared before use. This increase in flexibility is countered by the fact that programs cannot be checked for correctness until runtime.

A detailed overview giving examples of data types and describing what a type system consists of is given in chapter 3.

2.8 Conclusion

In this chapter we have described the two most popular distributed systems, DCOM and CORBA, with respect to their handling of complex types in heterogeneous environments. They have been compared to open coordination systems such as LINDA and it has been shown how their solution is not applicable in this environment. The LINDA coordination system has also been described and its handling of types has been detailed.

Two mobile agents systems, CLAIM and LIME, were described and it was shown that function types are different from mobile agents. A function type is more like a mobile component than a mobile agent because a running process can grab a

\(^9\)Although this will not determine problems such as non-termination, space leaks etc.
component (function) from the network, plug the component in and use it as if it had always been part of the same process.

Previous systems which have started from a LINDA base, or use a different model, fall into the same trap. They either ignore types and use a single language, restrict users to a small set of atomic types or use a convoluted third party type system in which the programmer must specify their types and have the code they will use generated from this specification. For systems which began by following the LINDA model this means that they can no longer be considered to be truly open LINDA systems, as they are either developed for a single language or the processes which form the network must share information before they are run. If the system limits the programmer to simple atomic types then the coordination primitives are no longer orthogonal to the host language, and this breaks a key facet of the LINDA model.

In subsequent chapters it will be described how types are represented in different programming languages and hardware architectures. It will then be shown how these different representations can be matched and converted between at runtime. This will then be integrated into an existing LINDA system while maintaining all key LINDA properties.
Chapter 3

Representations Of Types

Allowing processes to share types, in addition to sharing values, requires that there is a common specification which describes the types. In this chapter we will consider what data types should be supported, how they can be combined to form more complex types and how a new specification language can be used to describe them. Previous specification languages are used to give the definition of a type whereas in the proposed system two different processes may use the language to describe their own, possibly distinct, representations of a type. A representation of a type is an implementation that is used by the programmer. In systems such as CORBA the programmer defines a type and then creates a representation of that type using a tool to generate code. The following chapter will focus on the matching of two different representations of the same type. A process which infers the definition of the type from the different representations. This chapter will focus on the specification of a single representation of a type.

In chapter 2 XML type system and CORBA’s IDL were described. These languages give the definition of a type as it must be used everywhere in the system. For that to work the processes must have definitions prior to being run. These languages, and others like them, are not designed to be created from the description of a type in a program, rather the description in the programming language should be created from them. For this reason, and to clarify the design choices made as part of this work, a new language will be created which fits the requirements of the work (such as being able to represent types from any programming language).

By allowing types to be represented differently by different processes, the terminology used becomes important in order to avoid confusion about which aspect of the types and values are being discussed. In traditional systems the ‘type’ is something which is defined and which is used directly. In the proposed scheme a ‘type’ is never directly described, instead processes declare ‘type representations’: for example a type may be a pair of integers, while a process may declare its representation of this type to be a pair of 32 bit, big-endian bit strings. A particular instance of a type is known as value, for example \(\langle 1, 1 \rangle\) is a pair of integers. Just as types have representation so do values. A representation of the value \(\langle 1, 1 \rangle\) may be \(\langle 1, 1 \rangle\) where both the 1s are 32 bit, big endian integers. Typically the term ‘type
specification’ is used to mean the definition of a type, but in this thesis types are never directly described and instead are inferred by matching type representations. A type specification here means a specification which defines a representation of a type.

There are three classes of types which will be considered in this chapter.

**Atomic types** are types which are not constructed from other types.

**Compound types** are made up of atomic and compound types combined using a variety of operations.

**Native types** are types included as part of a programming language. They may be either atomic or compound types, but the programmer need not define them themselves. Types which are defined as part of a standard library may not need to be defined by the programmer to use them, but they have been defined by a programmer in the language being used, so they are not native to that language. In order to maintain a close integration with the host language, types which are included as part of the programming language should be made available by prespecifying compound types in the specification language. Those types which are defined in the standard library may be prespecified, but as they are not as common and can be specified by the programmer, this is not a requirement.

### 3.1 A Question Of Representation

Types can be divided into two groups, atomic and compound. Both can be represented as a (possibly infinite) set of values. However, there is a fixed number of atomic types, each of which has a number of well defined representations that cover different portions of the full range of the type. The atomic type ‘integer’ covers all integer values, from negative to positive infinity. A representation of this type may be 32-bit, big endian, two’s complement integer - a representation that can only represent values from -2,147,483,648 to 2,147,483,647. This contrasts with compound types, of which there are an infinite number, consisting of atomic and compound types combined using various operators.

Examples of atomic types are numbers, characters and booleans. Other types, such as complex numbers\(^1\) or strings, may be included as a native type in a programming language. As these types can be constructed out of the three given atomic types they are not themselves atomic. All commonly used programming languages separate integers from real numbers, and treat them as separate types. The range of values that can be represented by the integer type is a proper subset of the range of the real number type, which means that integers can be viewed as just another representation of numbers along with floating and fixed point numbers. Although

\(^1\)When considered as a pair of real numbers
languages tend to treat integers and floating point numbers as separate types internally, it is common for implicit conversion between the two types to be provided. This means integers can be treated as floating point numbers transparently, and vice versa. Processors internally work on integers and floating point numbers separately, as integers can be manipulated much more efficiently than floating point numbers. The separation at the programming language level therefore make sense, but as numbers will not be manipulated by a LINDA-like kernel the motivation behind this separation no longer holds.

Although the separation of integers and real numbers for performance reasons does not make sense in this environment, it does not change the fact that they are considered as separate types by programming languages. In languages where the integer type is separate there are two different ways the representation may differ. Firstly the order of the constituent bytes, in a multi-byte integer value, may be in a different order, or secondly the size of the representation of an integer value may be different. The abstract ‘integer’ type is an infinite set of values (which is a subset of values from the ‘numbers’ type), which cannot all be expressed in a concrete (finite) representation. The larger the representation the more values can be stored, but even when resorting to using a compound type to create an arbitrary precision integer, it is impossible in current computers to represent the complete integer type.

Types that are more complex than integers also have different possible representations, which are discussed in detail below. The answer to the question of which representation should be used is based on a number of factors. Representation of compound data types is affected by the representations used for atomic types, as well as the language and compiler used. For example the C language specification makes various requirements of the compiler in how they represent types in memory, compared to how they are defined in the program code, but some aspects are left up to the compiler writer. Other languages make different choices compared to C.

In the following sections, where the various ways of representing types are dissected, the discussion will focus on the abstract representation as well as two case studies for their concrete representations. The case studies will be C and Python. These two languages represent two very different language implementation strategies. C is ‘close to the metal’ while Python is a high level language running on top of a virtual machine.

The key reason for studying the many and varied ways a type can be represented is that in order to convert types between different languages and architectures, it is necessary to create an algorithm that can perform the conversion. That algorithm must be able to handle all possible representations. As the representation used for types is decided on a process-by-process basis, with no consultation between different processes, or between the processes and the LINDA kernel, any representation can be used and the kernel must be able to cope with them all.

---

2As all the LINDA does is store and retrieve values by matching
3.1.1 Inferring Specifications And Representations

This chapter is focused on how the programmer can specify the types they are going to use in a format the LINDA-like system can understand. It also deals with how values of those types are transferred to the LINDA kernel. In some languages the programmer already has to specify the types they wish to use. A programmer who is using C and wants to have a pair of an integer and a string must create the following type:

```c
struct IntString {
    int i;
    char* s;
}
```

If the programmer knows that the value that will be used in the integer portion of the type will always be between -128 and 127 then they may create this type:

```c
struct IntString2 {
    char i;
    char* s;
}
```

This is enough for the compiler to create an in-memory representation of the type, so why should the programmer have to specify the type again, this time in a new type specification language? The problem arises because different languages have different sets of built in types, and some languages have one type where another has two or more. C uses the 'char' type for both integers and characters. These are different atomic types which the C language designers chose to give the same representation. Python instead has the 'int' and 'str'\footnote{This type actually holds a string rather than a single character, but various functions including ord expect the string to hold a single character.} types. It is not possible from the type definition in the host language to know which interpretation of the type the programmer intended.

Other languages, such as Python, are dynamically typed and the programmer does not create complete type specifications. Instead they create values 'on the fly' using any types they like. Languages that do this are dynamic enough that it might seem like the values could be traversed at runtime and the type specification created directly from the values used. There are problems with this, for instance, as the types are never specified every value will need to be considered individually, meaning the specification generation algorithm would never infer a sum type. Sum types are a regular occurrence in programming and therefore the specification language must be able to handle them.

If a programmer creates a value \(\langle 1, "abc" \rangle\) a type specification can be inferred as

\[ x: \text{int}\star\text{string}; \]

What happens if the next value is \(\langle 1, 2 \rangle\)? The system will then infer the specification \( y: \text{int}\star\text{int}; \). If the system is inferring type specifications
on a value-by-value basis then it cannot know that the programmer wants both
values to be the same type (i.e. \( z : \text{int} \times (\text{string} + \text{int}) \)). If it were to infer
this then a value for each element of the sum type would need to be provided at the
same time. This would result in the programmer needing to specially create a series
of values to get the correct type to be inferred. This is would be a difficult and
error prone process, as it would be easy to miss a case, or for the algorithm to infer
the wrong type. By specifying the type representations themselves, the programmer
avoids any ambiguity.

The type specification for a list is another type which would cause problems
when inferred automatically. In the language proposed later in this chapter a list
of integers is \( \text{intlist} : \text{Nil} + (\text{int} \times \text{intlist}) \); but the value \(\langle 1,1,1,1 \rangle \) could
equally be interpreted as \( \text{intlist2} : \text{int} \times \text{int} \times \text{int} \times \text{int} \). One type spec-
ifies a variable length list while the other is of a fixed length. Which is meant is
only known to the programmer and both are valid interpretations of the value. An
algorithm could be designed to always infer one or the other, but as they both have
valid uses this would be unnecessarily restrictive for the programmer.

One possible solution is to combine automatic inferring of type specifications with
the use of additional information specified as part of the host language to guide the
inference. Even if specifications for type representations are inferred automatically
the work presented in this chapter would still need to be done, it would just be
hidden from the programmer. In making it explicit the programmer has complete
control over how the types that they use are presented to the system.

### 3.2 Atomic Types

Atomic types are the building blocks of any type system and are the smallest unit
of any value in the system. Typical classes of atomic data types include Booleans,
Numbers and Characters.

#### 3.2.1 Representation

These abstract classes of data types are not directly used in real programming lan-
guages because of the restrictions placed on the language by the computer hardware
they run on, instead subsets are defined and used as a native type by the language.
Booleans are often represented as a single byte where the following definition gives
the translation from representation to truth value.

\[
\text{bool} = \begin{cases}
false & 0, 1, \ldots, 255 \\
true & \end{cases}
\]  

(3.1)
There are a wide range of representations for an integer, below is a list of the most common ones.

\[
\begin{align*}
\text{sint8} & = -128, \ldots, 127 \\
\text{uint8} & = 0, \ldots, 255 \\
\text{sint16} & = -32768, \ldots, 32767 \\
\text{uint16} & = 0, \ldots, 65535 \\
\text{sint32} & = -2^{31}, \ldots, 2^{31} - 1 \\
\text{uint32} & = 0, \ldots, 2^{32} - 1 \\
\text{sint64} & = -2^{63}, \ldots, 2^{63} - 1 \\
\text{uint64} & = 0, \ldots, 2^{64} - 1
\end{align*}
\]

It is clear that the smaller types are subsets of the larger types, which in turn are also subsets of the full abstract integer type. The closest programming languages come to implementing the full integer type is using arbitrary precision integers (such as Python’s long type). These are typically made of multiple smaller integers with specially written mathematical operations to hide the implementation behind them.

Each of the representations listed above is in fact split into two separate representations, one for little endian systems and another for big endian systems. Integers are stored in bytes, and it is the order of the bytes that determine if the system is big or little endian. In a little endian system the first byte is the lowest value byte while the last is the highest. Big endian systems are the reverse. The pair of functions to convert between these representations form an isomorphism. Despite the fact that the Internet protocol standard [102] uses big endian as its default representation for integers [103] little endian numbers have no need to be converted. Conversion between the two representations is extremely efficient, and is the simplest example of how different representations of the same type can be isomorphic.

These types are all subsets of an abstract integer type, which is in turn a subset of the numbers type. Other representations of the numbers type are floating or fixed point types, both of which can represent a subset of the full range of the numbers type.

The Institute of Electrical and Electronics Engineers (IEEE) have produced a standard format [104] for representing real numbers using a floating point representation. The two most commonly implemented parts of the standard are single and double precision (given below) but there are a range of different possibilities with various sizes of exponents and fractions.

\[
\begin{align*}
\text{single precision (bits)} & = \begin{array}{c} \text{sign} \end{array} , \begin{array}{c} \text{exponent} \end{array}, \begin{array}{c} \text{mantissa} \end{array} \\
& = \begin{array}{c} 0, 1, \ldots, 9, 10, \ldots, 31 \end{array} \\
\text{double precision (bits)} & = \begin{array}{c} \text{sign} \end{array} , \begin{array}{c} \text{exponent} \end{array}, \begin{array}{c} \text{mantissa} \end{array} \\
& = \begin{array}{c} 0, 1, \ldots, 11, 12, \ldots, 63 \end{array}
\end{align*}
\]
The IEEE standard requires that all values are normalised, this means that all values are represented as \(1.a \times 10^b\) where \(a\) is the mantissa and \(b\) the exponent. The sign bit determines the sign of the whole value. In the exponent an additional bit is used to store the sign so for single precision values the exponent can be from -128 to 127. Single precision numbers have an accuracy of about seven decimal digits, while the double precision type has an accuracy of sixteen.

The floating point representations cannot represent the entire range of the real numbers, indeed many common values, such as 0.1, cannot be handled and must be approximated to the closest value that can be represented. In order to work around these problems real numbers can also be represented using fixed point notation. With floating point numbers the decimal point ‘floats’, meaning the number of digits before and after the decimal point changes depending on the magnitude of the value. Fixed point numbers do not alter the position of the decimal point. Fixed point values have the benefit of being able to represent every value within a given precision. There is no standard representation for fixed point values, and on modern machines, if they are supported at all, it is a language feature and not part of the hardware.

Different representations of integers are easy to categorise as they have higher or lower maximum and minimum representable values. Representations of real numbers differ in that although they also have upper and lower limits, different sets of values between these two limits are unable to be represented. Character data is usually treated as a different interpretation of numbers. However, the abstract character type is entirely different from numbers. The abstract character type is a set containing every textual symbol. Everything from simple ‘A’ to ‘z’ to the Cyrillic alphabet and the huge range of Chinese characters. Mathematical and punctuation symbols can also be classed as a character. As with numbers there are a selection of different representations, each capable of displaying a portion of possible characters.

American Standard Code for Information Interchange [105] (ASCII) is a simple 7-bit encoding of character data which can handle the Latin alphabet, numbers, some mathematical symbols and an assortment of other characters. Unfortunately, although both an American National Standards Institute (ANSI) and an International Standards Organization (ISO) standard \([105–107]\) exists there are various extensions to ASCII and country specific alterations to the supported characters. Extended Binary Coded Decimal Interchange Code (EBCDIC) is a character encoding similar to ASCII, except that it uses a full byte instead of just 7-bits. EBCDIC is no longer in mainstream use, apart from the IBM mainframes for which it was created. Those mainframes that do support it also support both ASCII and another character representation, Unicode, and have built in operating system support for converting between the three. Some later models, such as the IBM zSeries have processor instructions for aiding the conversion. Apart from the codes used to represent characters being different, it is identical to ASCII. As it is so closely related to ASCII and in such limited usage EBCDIC will not be considered further, but it
would not be difficult to add support for it to the proposed system.

One major criticism of ASCII and its derivatives is that it is focused on English speaking countries. Many symbols used in some countries were not possible to represent, and countries who use an entirely different alphabet could not use ASCII. The Unicode Consortium\(^4\) was formed to produce The Unicode Standard \([108]\) which has the aim of allowing the representation of every writing symbol in use today and with provisions to add more symbols as they are created.

Unicode provides a unique number for every symbol in its database, much the same as ASCII provides a number for each letter in the Latin alphabet. Each number is associated with a variety of visual representations, or *glyphs*. There are a number of different methods for specifying which Unicode character is required. These mappings include both variable and fixed width representations. UTF-8 \([109]\) is variable width, and can specify any symbol in the Unicode database. It uses between one and four bytes to represent each character, with the advantage that ASCII characters are represented using their original code, in a single byte. UTF-32 uses a fixed, four byte representation for characters. As all characters take up four bytes of space, UTF-32 is highly space inefficient if only the Latin alphabet is being used.

3.2.2 Equivalence

If the abstract atomic types were used in all programming languages then there would be no problem of equivalence between languages, as the types would all have the same range. As programming languages and computer hardware use subsets of the full types there may not be a mapping from all values in one language to all values in another.

The uint8 type, representing an unsigned eight bit integer is completely subsumed by the unit16 type, a sixteen bit unsigned integer. This means that any program (A) that uses the uint8 type can communicate with a program (B) that uses the unit16 type with no fear of data loss. The reverse is not true, B cannot communicate with A as A cannot represent any value over 255. Clearly this is a serious problem as they both use representations of the same abstract atomic type.

When determining the equivalence of atomic types it is reasonable to assume that all representations of the same abstract type should be treated as the same. Not all the values of a unit16 type can be converted into a value of the the uint8 type. Despite the fact that they are both representations of the same integer type, in this thesis uint8 and unit16 will not be matched. This is an instance of sub-typing, where one type can contain every value of another. In this work sub-typing will not be considered to focus on other aspects the type system. Future work may allow sub-typing to be used, and a clean solution to this problem found.

With floating point types the situation is not as clear cut as with integers. A fixed point type may be able to represent 0.1 but an IEEE floating point type

\(^4\)http://www.unicode.org
cannot. What should occur if 0.1 in a fixed point type is required to be converted into a floating point type? It is within the range and the precision of the floating point type, and can nearly be represented. Using a double precision floating point number the closest representable number to 0.1 is 0.1000000000000001. In many applications this would be close enough but in many it would not be. Rather than make assumptions about the numeric accuracy required by the user of the system, it is best to declare the value to be unconvertible and not take any chances with the accuracy of data being passed through the system.

Since boolean types only have two states, true or false, all representations must be able to represent both states. This means that unlike numbers all representations of booleans are equivalent.

The different representations of characters would be equivalent if they were all able to represent all written characters. Unicode, and its various mappings are able to do this, so it is possible to convert between them and maintain any character that is used. ASCII and EBCDIC only support a tiny fraction of the Unicode range and therefore, much like 8-bit integers compared to 32-bit integers, cannot represent all the possible characters. Unicode can represent all the characters in ASCII and EBCDIC so the relationship is a proper subset. Borrowing from the work above on the equivalence of different sized integers ASCII, EBCDIC and all the Unicode mappings are treated as unequal.

Formally, we define the equivalence of two atomic type representations, $t_1$ and $t_2$ as:

$$t_1 \simeq t_2 \iff \text{isatomic}(t_1) \land \text{isatomic}(t_2) \land \text{Values}(t_1) = \text{Values}(t_2)$$

where $\text{Values}$ returns a set of all possible values for the given type representation and $\text{isatomic}$ is a function which returns true if the given type is atomic, and false otherwise. The operator $\simeq$ is defined returning the equivalence of two types, subject to conversion using an isomorphism.

### 3.3 Compound Data Types

A compound data type is any type that combines atomic types or other Categorical Data Types (CDT) into a single type. In this section it will be described how compound data types are typically represented and how a type specification language should describe them. This section will not touch the issue of determining the equivalence of CDTs, which is discussed in the next chapter.

#### 3.3.1 Real language use of CDTs

Real programming languages often feature several different classes of compound data types, each with a different set of properties. These types can be considered to be either a product, sum or function type, or a combination of the three. A product contains multiple values, each of a specific type. A sum type contains a single value...
from one of a number of types while a function takes a value of one type, performs some process on it and returns a value of another, or perhaps the same, type.

In C or C++ a \texttt{struct} and a \texttt{class} are both instances of a product type containing data and functions. An object of a class type may have methods but it is unusual for the programmer to want the methods to be treated as data. These can easily be considered as external functions that act on the object, rather than functions that are contained within the object. The other form of a CDT that is built into C/C++ is the function pointer which is similar to a function type. The value of a function pointer is just a pointer to a function, rather than the function itself. Java is similar to C and C++ but without the \texttt{struct} construct. Java also does not have function pointers. C and C++ both define a \texttt{union} type which is similar to a sum type. A sum type contains a value of one of a number of types, but importantly it remembers what the type of that value is. The union type forgets the type of the value assigned to it and so is often combined with a struct in a pattern shown below.

\begin{verbatim}
struct IntFloatSum {
   enum {
      TYPE_INT;
      TYPE_FLOAT;
   } type;
   union {
      int i;
      float f;
   };
};
\end{verbatim}

Python has a wider range of built-in CDTs: tuples, lists and dictionaries as well as classes. A tuple can simply be represented as a product while a list is a combination of a sum and a product (as is shown below). A dictionary is a restricted function in the sense that it is a map from one set of values onto another, but it is a function which can be updated on-the-fly by the programmer.

There are too many languages to consider all of them here, but very few have built in support for types other than those in C, Python or Java. Prolog is one such language, but it is an unusual language in that everything in pure Prolog is built up from clauses. Clauses can have no parameters, in which case they are atomic values (terms) or they can have parameters in which case they act like products, or function calls. Prolog only distinguishes integers from other values, and then only in the case of some of its operators. Implementations of Prolog have extensions to make them more usable when implementing a complete program, such as input/output functions and string handling routines but these are not part of the core Prolog language.
3.3.2 Representation

Section 3.2.1 discussed how atomic types can be represented in memory. Compound data types, which combine atomic types to form new types obviously depend heavily on this. However, how their constituent values are put together also affects their representation.

Products are an ordered series of values that are treated as one value. C treats structures as single contiguous piece of memory, with the elements stored in the order that they are defined in the code [110]. The individual values inside the product may not follow directly on from each other as the C specification does allow for ‘pad bytes’. These can be added to the structure in order to give the memory layout the correct attributes for it to be efficiently accessed by the host CPU. Memory alignment is used by processors to enhance the speed at which they access memory. A 32-bit processor will often access memory quicker if it is aligned on a 4-byte boundary but the processor may not require this and may be able to access the memory even if it is unaligned. A compiler is unlikely to choose to make a program slower unless it has no choice and thus many types have pad bytes inserted. Other languages have different methods for representing products. Python does not use a contiguous block of memory and instead a product is a block of memory with pointers to the memory holding each of the elements. In C when a struct is copied all the values are copied as well. With Python only the pointers to the values are copied, so copying a product gives you two products with identical values. There are ways to copy the values as well (a deep copy) but it is not the default.

C is unusual in that it specifies how products are to be represented, but even it does not make guarantees beyond the fact that the values that make up the product will not be reordered. Many languages leave the representation of compound data types up to the compiler writer, and provided the compiler implements the interface specified by the language the programmer usually does not need to know the memory layout of their types.

C does not have a built in type which is equivalent to a sum type. The closest is the ‘union’ type which can hold values of a range of types, but it does not keep track of the type of the last value assigned to it. C’s ‘union’ is simply represented as a block of memory as large as the representation of the largest type it can hold. No additional storage is needed as it does not keep track of what type of value was assigned to the memory. If a proper sum type is to be implemented in C then it would consist of a structure containing an enumeration of the possible types, and a union of those types. The representation of a type such as this could be an integer followed by the usual representation of a union.

Python, like many scripting languages, allows any variable to hold a value of any type. This means that there is no need for a sum type as all variables are automatically a sum type which can hold any type. Programmers do not usually make full use of this and it is likely that the programmer will implement their own rules on what types are acceptable. A sum type allows the programmer to determine
what type a value held by a value of a sum type is, and in Python this is done through
the type function or _class_ member variable, which is common to all values in
Python.

Compiler options play an important role in determining the representation of
types. Taking the GNU Compiler Collection (GCC) C compiler as an example, one
of its command line options is `-mfaster-structs`. From the GCC documentation:
“With -mfaster-structs, the compiler assumes that structures should have 8 byte
alignment. This enables the use of pairs of “ldd” and “std” instructions for copies
in structure assignment, in place of twice as many ‘ld’ and ‘st’ pairs.” Compiler
options such as these can change the size of a type and although they will not
affect the program (apart from speed) they will have a catastrophic effect if a value
is transferred byte-by-byte to a process that expects the type to have a different
alignment.

Some languages leave the representation of compound types unspecified. Others
do not explicitly provide the three compound data types used in this thesis, or have
built in types not directly used by the language. This means that automatically
converting between the specification of the type in the language and the type spec-
ification language used here is impossible without modifying the existing language.
For this reason the programmer will be required to provide an isomorphism between
the value in the programming language, and the value as it will be sent across the
network. This will be studied in more depth once the type specification language
itself has been described.

### 3.4 Type Specification

It has been concluded that it is necessary to have a simple type specification language
that can be used to describe types from a wide range of languages, running on a wide
range of hardware. In some languages it is possible to infer the type of a variable,
while in others it is part of the program code. Systems such as CORBA use an
external language - the IDL - to specify the types. Although not a new idea the
language specified here is different from most others in that it is designed to represent
types from any language. Indeed, it must be general enough to specify types from
the majority of languages, and to describe types already created by the programmer,
rather than being used to generate code for types from the specification.

Type specifications are given an alpha-numeric string which serves as the types
name and this allows the specification to be referenced by another type, or itself.
Type names are only used within a single process, and are only needed to implement
recursive types, or to split large specifications into more manageable, smaller pieces.
The specification is either a type name, a sum, a product or a function. Type names
can either be another type or a predefined atomic type. Although type matching
is not based on the names of types it is essential that the types are given names to
allow recursive types to be defined.

There can be many specifications in a single piece of text, and they need not be
in any particular order. Type names which are contained within type specifications
are looked up as needed so types that refer to each other need not be specified in
order. It also allows for mutually recursive type definitions to be easily implemented.

Good code is usually platform independent which means that the same code
can be compiled for two different platforms and will run on either. It is therefore
important that the type specifications are also platform independent. Platform
independence in this case means that type sizes, formats and byte orderings are not
specified and are instead derived based on where the code is compiled. In order for
this to be possible the atomic types available in the type specification must match
those that are available in the host programming language. A program written in
C should be able to specify atomic types of \texttt{int} and \texttt{char*} as these will be most
natural to a C programmer. A Python programmer will expect to be able to use
\texttt{int} and \texttt{str}.

The binding code that gives each programming language the ability to create
type specifications defines a number of types, one for each of the types defined by
the programming language. A C binding which is compiled on a 32 bit machine may
create a type \texttt{long :: int32}; while the same binding on a 64 bit machine will
define the type \texttt{long :: int64}. Type names such as \texttt{int32}, \texttt{int64}, \texttt{ieesingle},
\texttt{ieeedouble}, \texttt{ascii}, \texttt{utf8} and \texttt{utf16} are just some of those that are declared as
reserved. They are used to represent atomic types that are used by the hardware.
The bindings create types which map from the programming language name to the
system’s name for the type as used by the hardware.

The product type is specified using * as the operator. A number of types are
joined together using the star to form a product type. The operator is left associative
but brackets can be used (and are recommended) when a product has complex types
as elements, in order to ensure the type is defined as the programmer intended.

Sum types are specified in a similar fashion to the product type, except with +
as the operator. Sum types have a lower binding priority than product types.
Some languages provide built in types that are not atomic types, and that can be expressed in the type specification language. Many languages provide a ‘string’ type but this can be expressed as a list of characters. As part of a ‘standard library’ a string type is defined as \texttt{string: Nil+(ascii*string)}; There are types similarly defined for UTF-8 strings, and other encodings. This can be used by the programmer as if it were an atomic type, but internally it is treated no differently from any other compound type.

Languages that provide more complicated base types, such as lists or dictionaries, pose a problem. One of the goals of this language is that it is simple. The proposed language could be extended through the inclusion of ‘type functions’ (functions that operate on types rather than values). For example the function \texttt{list} could be used such that \texttt{intlist: list(int)}; would be equivalent to \texttt{intlist: Nil+(int*intlist)}; This would complicate the language but requiring the programmer to create the full type specification is not a large hurdle for them to overcome. Therefore, the ability to define functions that operate on types which would allow the type specification language to be further integrated with the type system of the host language is left as future work. Only atomic types will be included in the language and, with the exception of the extremely common ‘string’, no other compound types are defined automatically.

Function types are represented in the specification language as a pair of types separated by a \texttt{->} symbol. The type on the left of the arrow is the parameter to the function, while that on the right is the return value. Multiple parameters are handled through the use of currying, in the same style as Haskell. No explicit support is given for multiple return values, but a product type can be used to achieve the same effect.

The final type is the pointer, which takes another type name as a parameter. A pointer to an integer is expressed as \texttt{p :: ptr(int)}.

Before values can match the specification, the specification itself must be valid. The client program parses the specification into an intermediate format (in the reference implementation developed in this thesis, this is XML) using the grammar given above. This intermediate format is the one sent to the server, rather than the original specification. The intermediate XML format could be validated against an XML schema to ensure the validity of it, but for the purposes of this work it is assumed that once initially parsed all specifications are valid.

### 3.4.1 Building And Converting Types

In this section the type specification language described above will be used to create specifications for a number of compound data types in different languages. The code required to convert values of those types to and from a representation suitable for transfer will be shown.

The first type to be considered is a pair of integers. In C this type can be specified thus:
struct IntPair {
    int a;
    int b;
};

Using the type specification language this becomes IntPair: int*int;. The pair of functions that must be created by the programmer are given below.

/* Allow the IntPair type to be serialised. */
LindaValue IntPairToValue(void* pair, LindaMemo memo) {
    /* Create a two element product type,
       then fill the two values with p->a and p->b. */
    LindaValue v = Linda_product(2);
    struct IntPair* p = (struct IntPair*)pair;
    Linda_setProduct(v, 0, Linda_toInt(p->a));
    Linda_setProduct(v, 1, Linda_toInt(p->b));
    return v;
}

/* Unserialise a value into an IntPair value. */
void* ValueToIntPair(LindaValue v, LindaMemo memo) {
    /* Allocate the memory for our value. */
    struct IntPair* pair = malloc(sizeof(struct IntPair));
    /* Register our memory in case another value is
       a pointer to it. */
    Linda_memoRegister(memo, v, (void*)pair);
    /* Get the two elements of the product, extract
       the integers and set the elements of the pair. */
    pair->a = Linda_getInt(Linda_getProduct(v, 0));
    pair->b = Linda_getInt(Linda_getProduct(v, 1));
    return pair;
}

Unfortunately the code given above has to work around C’s type system by using void pointers to carry values of arbitrary types. The first of the two functions converts a value from C’s IntPair type to the Linda representation of the same type. The second function performs the reverse operation. The purpose of the Linda_memo parameter and the Linda_memoRegister function will be explained below. The first of the two functions creates a two element product then sets the two elements of the product to an integer created from the appropriate element of the structure. The reverse function creates a new value to hold the structure and then sets the a and b elements to values from the product.

The equivalent pair in Python would be represented using a class. The two conversion functions given after the class definition are the Python equivalents of
the C functions given above.

class IntPair:
    def __init__(self, a, b):
        self.a = a
        self.b = b

def IntPairToValue(pair, memo):
    return linda.Value((pair.a, pair.b))

def ValueToIntPair(v, memo):
    pair = IntPair(int(v[0]), int(v[1]))
    memo.register(v, pair)
    return pair

Python’s dynamic type system and object oriented features make the two functions much simpler than their C counterparts, but their format is very similar. In particular the linda.Value function is able to convert a Python tuple into a LINDA product value, and Python integers into the equivalent LINDA integers.

The next type we will consider is a tree of integers. The type uses pointers to hold the branches of each node. These branches can either contain another node, or be empty. If both are empty the node is a leaf of the tree.

struct IntTree {
    struct IntTree* left;
    int value;
    struct IntTree* right;
};

In Python the same type is defined as:

class IntTree:
    def __init__(self, value, left = None, right = None):
        self.left = left
        self.value = value
        self.right = right

Both these types can be represented by IntTree :: ptr(IntTree) * int * ptr(IntTree));. The pair of functions for the C language is given below.
/* Serialise an IntTree value. */
LindaValue IntTree2Value(void* tree, LindaMemo memo) {
    /* Create the three element product. */
    LindaValue v = Linda_product(3);
    struct IntTree* t = (struct IntTree*)tree;
    /* Register this value with the memo in case a pointer points back to us. */
    Linda_memoRegister(memo, v, tree);
    /* Set the first element of the product to a pointer to left branch. */
    Linda_setProduct(v, 0,
        Linda_ptr(Linda_getValueFromMemo(memo,
            IntTreeType, t->left)));
    Linda_setProduct(v, 1, Linda_getInt(t->value));
    /* Set the third element of the product to a pointer to right branch. */
    Linda_setProduct(v, 2,
        Linda_ptr(Linda_getValueFromMemo(memo,
            IntTreeType, t->right)));
}

/* Deserialise a Value into an IntTree */
void* Value2IntTree(LindaValue v, LindaMemo memo) {
    /* Create space for the tree. */
    struct IntTree* tree = malloc(sizeof(struct IntTree));
    Linda_memoRegister(memo, v, tree);
    /* Extract the first element of the product, get the value being pointed to and ask the system to convert it into an IntTree automatically. */
    tree->left = Linda_getRealFromMemo(memo, IntTreeType,
        Linda_getPtr(Linda_getProduct(v, 0)));
    tree->value = Linda_toInt(Linda_getProduct(v, 1));
    /* Extract the third element of product, get the value being pointed to and ask the system to convert it into an IntTree automatically. */
    tree->right = Linda_getRealFromMemo(memo, IntTreeType,
        Linda_getPtr(Linda_getProduct(v, 2)));
    return tree;
}

In these two functions the ‘memo’ parameters are used to associate values with the pointer it was calculated from, or when converting back it associates pointers to values. The three key functions are Linda_memoRegister, Linda_getValueFromMemo
and \texttt{Linda\_getRealFromMemo}. \texttt{memoRegister} behaves differently depending upon which direction the conversion is being performed. If a value is being serialised then it associates the value with the pointer that is passed as a parameter. If a value is being de-serialised then it associates the pointer with the value. The two other functions return elements from the ‘memo’ structure, or call the appropriate function to create them if they do not exist. The purpose of the ‘memo’ is to convert pointers from the machine specific value into a general value capable of being transmitted, and back again. It is important that the programmer registers their value with the ‘memo’ before calling the retrieval functions in order to allow circular data structures to be handled.

The functions described above are called once for each individual value that makes up the entire data structure. A cyclic value may consist of two values, each of which contains a pointer to the other. In order to be able to serialise this kind of value the system must know which values have already been converted, and what they were converted into. When the system is asked to convert a value that it has already converted it can detect this, return the already converted value and break the cycle. Without the ‘memo’ structure serialising cyclic values would be impossible.

### 3.4.2 Example Specifications

Python has built in support for lists and these lists can take any types as their elements. If the list only contains integers then the specification in 3.13 can be used.

\[[1, 2, 3, 4] \to \text{intList} : \text{Nil} + (\text{int} * \text{intList}); \quad (3.13)\]

If the Python list contains values of types which alternate between integers and strings then the specification in 3.14 will represent it.

\[[1, "a", 2, "b"] \to \begin{cases} 
\text{intList} : \text{Nil} + (\text{int} * \text{stringList}); \\
\text{stringList} : \text{Nil} + (\text{string} * \text{intList}); 
\end{cases} \quad (3.14)\]

A Python dictionary can also hold any value, if we assume the keys are strings and the values are integers the type specification is...

\[\{"a" : 1, "b" : 2\} \to \text{strDict} : \text{string} \to \text{int}; \quad (3.15)\]

This section has shown how a small subset of the possible types can be represented in the type specification language and used in real programming languages. The atomic types not shown follow the same pattern as for integers, while sum types have no special functions as they consist of a single value of one of their constituent types. The details of representing function types have not been mentioned, and are dealt with in chapter 5.
3.5 Conclusion

This chapter has discussed how types are represented at a low-level and how compound data types are built up from the simple atomic types. A simple language for describing these types has been proposed. It specifies types in a form compatible with the type isomorphisms which will be presented in the next chapter. The type specification language, when combined with a type isomorphism algorithm will provide a flexible method for matching equivalent types and transferring values between heterogeneous systems.

The implementation of the serialisation mechanism was outlined and its use in two languages, with very different type systems, was demonstrated. The difficulties in performing the serialisation of values automatically were described as well as the problems with automatically determining the type specification to be used. Because the type specification language proposed does not directly support all types supported by all programming languages the programmer must create a pair of functions to convert types from their representation to the representation they wish to expose to the system. The functions are associated with the type value when the specification is parsed so that they can be called automatically, as needed by the system.

The work in this chapter showed how typed values can be serialised partially automatically. The following chapters will show how the types created in this chapter can be matched, and how this can be integrated into Linda.
Chapter 4

Type Isomorphisms

In the previous chapter it was shown how a type may have many different representations. Different processes in a LINDA system use different representations of the same type. This type is never specified, and instead its existence must be inferred from the fact that the different representations are isomorphic. In this chapter various isomorphism algorithms, for determining whether two representations are representations of the same type, will be discussed.

Conversion functions that take a value of one representation and convert it into another allow different representations of the same type to be considered equivalent. This means that processes that use different, but equivalent, representations will be able to communicate without knowing that the other process is using a different representation. Probably the biggest reason for different representations of the same type existing is that they are created in different programming languages. Allowing programmers the opportunity to create parts of their system in a language most suitable for the task will make their life easier. Other systems that generate types in a programming language using a third-party type specification language are unlikely to produce a type as it would have been designed by a human programmer across all languages the generator supports; recall the example in section 2.3 of CORBA mapping onto Lisp. If the programmers themselves can write the specification for the type, and the system automatically determines which type representations are equivalent, then the type will be ‘natural’ in all languages.

There is a concept that will allow an equivalence class of types to be defined, and will provide conversion functions between the elements of that class — a ‘type isomorphism’. In this chapter some of the many and varied algorithms for inferring type isomorphisms will be discussed. Many of these algorithms were created with the aim of searching for types in libraries. This means that they are not suitable for use in a system where a user can have no input to the algorithm at runtime. The method they use to generate the isomorphism is still useful and one example algorithm will be given. The exact requirements for a suitable algorithm will be spelt out and such an algorithm, which incorporates elements from previous algorithms, will be detailed.
4.1 What Are Type Isomorphisms?

A better first question would be what is an isomorphism? Below is the dictionary [111] definition of an isomorphism.

**isomorphism n.**
- Biology. Similarity in form, as in different generations of the same life cycle.
- Chemistry. The existence of two or more substances of different composition in a similar crystalline form.
- Mathematics. A one-to-one correspondence between the elements of two sets such as those of Arabic and Roman numerals.

Clearly the third definition is the closest to the subject of type theory. Recall that a type can be thought of as a (possibly infinite) set of values. When applied to this interpretation of a type the definition becomes that there is a one-to-one correspondence between the values in the sets of each type. The correspondence is in the form of a pair of functions which convert values between the two sets and which are inverses of each other. As an example consider two isomorphic types, $A$ and $B$. An isomorphism between them will consist of two functions $f_A :: A \rightarrow B$ and $g_B :: B \rightarrow A$ such that $\forall a \in A. a = g(f(a))$ and $\forall b \in B. b = f(g(b))$.

A type isomorphism algorithm is a mechanical process that takes two type definitions and returns the pair of functions that make up the isomorphism, if it exists. The algorithm is typically a recursive function that has a rule for each individual class of type (atomic, sum, product etc.). Three such algorithms are described in section 4.2. However, next we will describe what the rules must be for types to be considered equivalent. In type literature creating a type isomorphism is usually described as ‘inferring’. The word inferring has connotations of manual work (apart perhaps from inferring the types of variables, as in the Hindley-Milner algorithm [112]), and in some sense implies that the result was a guess (albeit an educated one). Consequently, when the use of an isomorphism algorithm is being described the word ‘generated’ will be used. As the algorithm is being mechanically followed with no user interaction ‘generated’ fits the action much more closely than ‘inferred’.

4.1.1 Type Equivalence

The rules which govern whether two types are equivalent or not are largely dependent on what the generated isomorphism will be used for. The rules can be defined to classify a wide range of types as equivalent, or to give a very restricted set. There is no universally accepted definition of ‘equivalent types’, instead each system has its own definition. There is some crossover between these definitions and that is what will be described here.
Atomic data types are the simplest of all data types and consequently they also have the simplest rules for equivalence. If two types representations, $A$ and $B$, are identical then they are equivalent. This was discussed previously in chapter 3, and it was decided that although 16 and 32 bit integers are both representations of the same abstract type, an integer, they have different ranges of values and therefore cannot be considered equivalent. Later in this chapter a Galois [113] connection will be considered rather than an isomorphism, which opens up the possibility of two different atomic type representations being equivalent.

**Product Types**

Product types contain a number of other types in an ordered collection. A type $C$ may contain two other types $A$ and $B$. A product type also consists of a constructor function which creates a value of that type and a set of accessors which return the values used to create the value. If the constructor function is $\times$ then $a \times b \rightarrow c$ where $a \in A$, $b \in B$ and $c \in C$, and the accessor functions will be defined as $f_A(c) \rightarrow a$ and $f_B(c) \rightarrow b$ such that $\forall c \in C. f_A(c) \times f_B(c) \equiv c$. The product type can be implemented as either $A \times B$ or $B \times A$. If the constructor and accessor functions are defined appropriately then to the outside user the two representations will appear to be identical. Using the constructor/accessor method for describing the representation of a product type makes it easy to decide whether the two representations are isomorphic, but constructing the isomorphism will involve composing and evaluating the functions. Specifying a representation of a type in this way requires the programmer not only to specify the functions so they can be evaluated, but also the semantics of the functions so the isomorphism algorithm can work with them.

Instead of specifying the constructor and accessor functions, and using them to generate the isomorphisms the programmer can simply specify the logical view of the data that they have. The constructor and accessor functions are implicit when specifying products in this way, and so do not need to be specified separately. As values need to be serialised by a client process before sending them to the server this serialisation can easily involve reordering the data if the physical representation of a type is different to its logical representation.

To match a product type the type representation of each element must be isomorphic to the type representation of the equivalent element in the other product type. Some isomorphism algorithms allow the elements of product types to be re-ordered, or even for them to not have the same number of elements (though in this case they do not form an isomorphism). As the requirement has been made that the physical ordering of a value of a product (when it is presented to the isomorphism algorithm) must be the same as the logical ordering, the elements of the product types cannot be reordered. In addition as values may be converted in either direction the functions created by the algorithm must be a true isomorphism and therefore the two representations of a product type must have the same number of elements.
\[ t_1 \simeq t_2 \iff \text{isproduct}(t_1) \land \text{isproduct}(t_2) \land \text{len}(t_1) = \text{len}(t_2) \land \forall i \in 0 \cdots \text{len}(t_1). t_1[i] \simeq t_2[i] \]

The above equation defines the equivalence for two product type representations, \( t_1 \) and \( t_2 \) where \( \text{len} \) returns the number of elements in the given type, and the \([n]\) operator returns the \(n^{th}\) element of the type. The function \( \text{isproduct} \) returns true if the given type is a product, and false otherwise.

**Sum Types**

A value of a sum type contains only a single element, but that element can be of a variety of types. It is clear that for two sum types to be isomorphic it should be possible for all their constituent types to divided into pairs, with each type providing one half of the pair, and for all the pairs to contain isomorphic types. There are a number of issues with this, such as what happens when elements of the same type are isomorphic to each other. Imagine a representation of a type, \( A \), defined as \( B + C + D \) where type representations \( B \) and \( C \) are isomorphic to each other. Now assume \( A \) is being compared to another type representation, \( X \), defined as \( Y + Z \) where \( Y \) is isomorphic to \( B \) (and by extension to \( C \)) while \( Z \) is isomorphic to \( D \). All values of the type representation \( A \) have an equivalent value of type \( X \), and vice-versa, therefore the two types are isomorphic. The problem is that some values of type representation \( B \) have more than one value of type representation \( A \). If a value of type representation \( X \) contains a value of type representation \( Y \) then this can be converted into a value of either type representation \( B \) or type representation \( C \). Although \( B \) and \( C \) are isomorphic they may be distinct representations.

Having two isomorphic representations as part of the same sum type is unusual, but is perfectly valid. The ‘no user interaction’ rule imposes a large restriction on possible solutions to this problem. The first solution is to simply disallow types that are isomorphic to each other from being included as part of the same sum type. This is unacceptable, as it imposes a very strict condition on the types that are allowed by the system, contradicting the aim to provide the most flexible type system possible. The other possibility is to only match sum types where each element in the sum type has a counterpart in the type it is being compared against. If types in a sum type are isomorphic to each other then for that type be be declared equivalent there must be a counterpart for each element in the type it is being compared against. As values of any type in the group of isomorphic types can be converted to any other type in that group an arbitrary ordering must be imposed to define which type the value will be converted to. This is expressed in equation 4.1.
 Comparisons inside a tuplespace as part of a Linda system can be divided into two groups — either a value is compared to a representation of a type (matching against a formal), or it is compared to another value (matching against an actual). Consider two sum types, $A :: B + C$ and $X :: Y + Z$ where $B, C, Y$ and $Z$ are all isomorphic. Clearly by our definition above $A$ is isomorphic to $X$ but there is an element of uncertainty in the conversion between values of the two representations. If the comparison is between two values then there is no confusion as the value that is returned to the program that made the request is the value it sent. If a value of one type is being matched against a representation of a type, for example if a value of type $A$ where the element is of type $B$ is compared against $X$ then the value could be converted to both $Y$ and $Z$ as they are both isomorphic to $B$. Since there is no further information available to the isomorphism algorithm it must make an arbitrary decision, and in this work sum types will be converted to the equivalent isomorphic type in the left-right ordering (considering only the isomorphic types) as defined in the textual specification of the type. This forces values of type $B$ to be converted to type $Y$, while $C$ values will be turned into values of type $Z$. Any ordering, even a random one, would be valid but by specifying it as a static ordering, the programmer gains greater understanding of why values were returned as a particular type.

**Function Types**

Function types contain only two other types, the parameter and the result. Functions which take more than one parameter can be turned into a series of functions which take only one through the process of currying.\(^1\) Two functions are equivalent if and only if the parameter types of the two functions are isomorphic, as well as the return types. This definition is simple, but deriving conversion function for the functions is considerably harder and is dealt with in a separate chapter (see chapter 5).

The equation given below defines the equivalence relation for function types. The function $isfunction$ returns true if the given type is a function, and false otherwise. The functions $param$ and $return$ return the parameter and return types of the given function type respectively.

\[
\begin{align*}
  t_1 \simeq t_2 & \iff isfunction(t_1) \land isfunction(t_2) \land param(t_1) \simeq param(t_2) \land \\
  & \land return(t_1) \simeq return(t_2)
\end{align*}
\]

\(^1\) $A \times B \rightarrow C = A \rightarrow (B \rightarrow C)$
The work in this thesis does not deal with subtyping, but if it were to, then
function types pose an interesting problem. As functions execute on values, and
return a value the matching between two function types must compare the argument
types in one direction and the return types in the other. Take two functions, \( f \) and
\( g \), and their arguments \( f_a \) and \( g_a \) and return types \( f_r \) and \( g_r \). When matching, \( g_a \)
must be a subtype of \( f_a \) while \( f_r \) must be a subtype of \( g_r \). The covariance of the
argument types and contra-variance of the result types is due to the direction in
which the values are passed ‘through’ the function. As subtyping is not considered,
and all matching is done using isomorphisms the covariance/contra-variance of a
function’s arguments and return type is not an issue.

4.1.2 Pointers

One type that is not mentioned in isomorphism literature is the pointer. Pointers
are used in many languages, and in some languages, such as Java, all variables hold
a pointer to a value rather than the value itself. This means that pointers are not
explicit in the language but are instead implicitly used across the entire language.
By not considering pointers as a type most type isomorphism algorithms severely
limit their applicability to real languages.

Support for pointers is required in a type system to allow cyclic types. A pointer
to a type \( A \) can, in some respects, be considered as \( Nil + A \) where \( Nil \) is a special
value indicating that no value is being pointed to and \( + \) is the usual sum type
operator. This definition does not take into account the fact that the pointer may
point to another value, or even itself. Were \( Nil + A \) to be isomorphic with \( \text{ptr}(A) \)
then it a cyclic value would have to be converted into a linear value. This is clearly
impossible and as the type system proposed in this thesis allows programmers to
create values that require the use of pointers the type isomorphism algorithm must
also support them. As pointers ‘point’ to a value of a specified type it is obvious
that a pointer to a type \( A \) is isomorphic to a pointer of type \( B \) if, and only if, \( A \) is
isomorphic to \( B \).

As with the other types the equation below defines the equivalence relation for
pointers. The function \( \text{ispointer} \) returns true if the given type is a pointer, \( \text{type} \)
returns the type of the value pointed to by the type.

\[
t_1 \simeq t_2 \iff \text{ispointer}(t_1) \land \text{ispointer}(t_2) \land \text{type}(t_1) \simeq \text{type}(t_2)
\]  

(4.1)

A naive implementation of an isomorphism algorithm would loop indefinitely
while comparing \( A : \text{int} * \text{ptr}(A) \) and \( B : \text{int} * \text{ptr}(B) \), which are both linked lists
of integers. To allow types such as these to be matched the isomorphism algorithm
should treat types as separate entities and not compare all types as one unit. The
algorithm simply needs to keep a list of which types are currently being compared.
If the algorithm is asked to compare two types that it is already comparing then
it can assume they match. If they are actually not equivalent then the algorithm will still determine them to be not equivalent, as expected, later in the algorithm’s execution.

Part of the role of the type isomorphism algorithm is to create functions which will convert between values. In the case of a pointer this means calling the function created by the isomorphism algorithm between the types pointed to by the pointers, storing them in memory and replacing the value with the new memory address. This has already been discussed, in the previous chapter, with regard to serialising cyclic types.

4.1.3 Use Of Type Isomorphisms

A major use of type isomorphisms has been in searching program libraries [114,115]. The programmer specifies the type of a function or object that they wish to use; the system then scans a library for types that are isomorphic to the query. In some cases the system will just return a list of matching types, possibly ordered with some scoring system or it may return a function which wraps the matching functions and gives it the appearance of having exactly the same type as that requested by the programmer.

Isomorphism algorithms designed for searching are typically very general and allow a wide range of types to be isomorphic to each other. As the searching tool is an aid to the programmer it is better that it returns a list of functions rather than none at all. The ideal, of course, is that it returns the type the programmer is looking for, and only that type, but with many functions in a library typically sharing the same or very similar types this is unrealistic.

If an isomorphism algorithm matches more than one type user intervention is required to select which of the matched types is the correct one. The situation in which type isomorphisms are being used for this thesis is quite different. The programmer provides a type specification which is compared, one-by-one, to the types currently in use by values in a tuplespace. As was stated when the programming interface to tuplespaces was defined, if more than one tuple matches the template then one is returned non-deterministically. If more than one tuple has all of the types of its elements matched by the algorithm then one is selected and the values are converted using the generated isomorphism and the resulting tuple is returned. All this takes place inside the LINDA kernel, where no user interaction can occur.

4.1.4 Galois Connection

A Galois connection [113] is a close, although weaker, relative of the isomorphism. Every Galois connection gives rise to an isomorphism between certain sub-posets of the sets which the connection is between. A Galois connection is a pair of functions between two posets such that for the sets \((A, \leq_A)\) and \((B, \leq_B)\) the functions \(F\) and \(G\) are defined as \(F : A \rightarrow B\) and \(G : B \rightarrow A\). \(F\) and \(G\) have the property that \(F(a) \leq_B b\) if and only if \(a \leq_A G(b)\).
Galois connections are often used as part of the abstract interpretation of programming languages, for example a Galois connection can be formed between the source code of a program and the compiled object code. The functions that make up the Galois connection consist of the compiler and a disassembler. Because of optimisations and other manipulations that the compiler performs the disassembler may not return the same source code as was compiled, but it must perform at least what the original did.

The Galois connection allows the value which has been passed through both functions to be different from the value it was initially, but the ordering must be maintained. This is analogous to the issue with different sized representations of atomic types, which was discussed in section 3.2.2. The abstract atomic types are represented by types which have a limit on the values which they can hold. If the type representations $A$ and $B$ are both representations of the same abstract type but $A$ is a ‘smaller’ representation, i.e. $B$ holds all the values that can be held by $A$ and more. It is possible to create a Galois connection between these two sets of values as for all values in $B$ greater than the ‘largest’ value in $A$ the value is converted to that value when converted from $B$ to $A$.

As a specific example of why a Galois connection rather than an isomorphism applies to the conversion between representations of atomic types, consider an 8-bit unsigned integer and a 32-bit unsigned integer. The conversion function between the `uint8` and `uint32` simply extends the value by adding bytes containing the value 0. For values less than 256 the reverse functions truncate the extended value and return the original value, all other values will be returned as 256. With the appropriate definitions for the ordering of the posets this constitutes a Galois connection.

When communicating through a LINDA system the programmer can reasonably expect that their values will be unchanged when they are eventually retrieved from a tuplespace. Unchanged in this case means that the value can be converted back to its original format and it will be identical. This is not possible to guarantee with a Galois Connection as converting a value from a ‘large’ representation to a smaller one will lose information.

In a system which supports subtyping, a Galois Connection may be the only solution for the type matching puzzle. As the system proposed in this thesis does not, and as using a Galois connection would cause some values to be irreconcilably altered as they passed between processes, type isomorphisms will be used. This ensures that values that can be represented by both processes will be treated as expected, and values that cannot be represented by the destination process will never be transmitted to that process.

### 4.2 The Algorithms

In this section we will detail three algorithms for inferring type isomorphisms. The first is the simplest possible algorithm as it only matches identical representations of the same type. This algorithm is provided as a basis for comparison to the other
two algorithms which relax their definition of an equivalent type.

4.2.1 Structural Comparison

This algorithm is a straight comparison of the structure of the two types. This algorithm is the ‘null’ algorithm for inferring type isomorphisms as the definition of type equality is as restrictive as possible (without declaring some identical types to be not equivalent). This algorithm has not been published due to its simplicity but it is included here in order to provide a base to compare the other algorithms against.

The algorithm below refers to two type representations, A and B. Where two type representations are being compared == represents an application of this algorithm to the two types. Where two abstract types are being compared (as in type(A) == Atomic) == represents a direct equality test. The algorithm makes use of three functions. Firstly type returns the abstract type of a type representation. For example, for a type representation x :: y * z; the function call type(x) will return Product. The final two functions work on function types only. parameter returns the parameter type of the function, while return gives the type representation returned by the function.

```python
def compare(A, B):
    if type(A) != type(B):
        return False
    elif type(A) == Atomic:
        return A == B
    elif type(A) == Product:
        foreach a in A, b in B:
            if not (a == b):
                return False
        return True
    elif type(A) == Sum:
        foreach a in A, b in B:
            if not a == b:
                return False
        return True
    elif type(A) == Function:
        if not parameter(A) == parameter(B):
            return False
        elif not return(A) == return(B):
            return False
    else:
        return True
```

The algorithm pattern matches against the abstract types of the two type repre-
sentations being matched. If they are both atomic types then it compares the types using an equality test. Product types are compared pairwise, using repeated applications of the algorithm to each pair of types. Sum types are matched by finding a mapping between the consistent types of both \( A \) and \( B \) such that the pairs are all isomorphic. Finally function types are compared by comparing the parameters and the return types.

This algorithm does not reorder product types so in its definition of type equality products are not commutative. This means that product types are treated more like tuples than like C structures where the order of the elements is irrelevant.

The structural algorithm handles functions, which the other two algorithms do not in the form that they are initially defined. Functions are equivalent if their parameters and return types are equivalent. This is exactly the requirement spelt out above.

4.2.2 Di Cosmo 2003

In the 2003 paper [116] Roberto Di Cosmo, Didier Remy and Francois Pottier describe a tool that uses type isomorphisms to retrieve classes from object-oriented libraries. The tool builds the necessary glue code such that the class which is returned exactly matches the specification being searched for, not the class that is matched. A user may search for a class that has two methods, \( x \) and \( y \). If the algorithm finds a class that has methods \( x, y \) and \( z \) then a ‘wrapper’ class will be created to mask method \( z \). It will also create new methods if the matched class has isomorphic but not identical type signatures for its methods.

For example:

```java
interface I1 {
    float m1 (I1 a);
    int m2 (I2 a);
}
interface I2 {
    I1 m3 (float a);
    I2 m4 (float a);
}
```

The two Java interfaces given above may be encoded (ignoring the names of the methods) as the mutually recursive types \( I_1 = (I_1 \rightarrow \text{float}) \times (I_2 \rightarrow \text{int}) \) and \( I_2 = (\text{float} \rightarrow I_1) \times (\text{float} \rightarrow I_2) \). The Di Cosmo algorithm takes a list of types such as these and finds classes that match the specification given by the user.

In the application that is the focus of the paper, searching a library for classes matching a given type specification, testing for equality of the two types is not the right relation to use. The actual interface may have many methods and a complex type signature so it is not feasible to ask a user to correctly guess the type specification for all methods in the class’ interface. For this reason Di Cosmo
et al. introduce the notion of AC-subtyping\(^2\) to relax the restriction on matching types. This matches interfaces which are a subtype of the specified interface, i.e. they match if they have all the methods and (perhaps) some additional ones too. Matching up to AC-equality will only match types where the whole of the type specification can be matched (although as it is commutative the order need not be correct). AC-subtyping will return types where the specification is a super-type of the found type. This allows the user to specify only a few of the methods of a class and the algorithm will still be able to match it correctly.

The arguments in favour of introducing AC-subtyping make perfect sense in the situation for which the algorithm was originally designed. However, although it is possible for the algorithm to throw away information when converting a value in one direction between values of two types, it is not possible to create it when converting the other way. When the tool is used to search a library it is always the type that is being searched for that is the smaller and the type which is matched that is larger. Once matched the type which was searched for is the type that will actually be used by the programmer. Values of the type being searched for (i.e. the smaller type) will never be converted into a value of the matched type (i.e. the larger type) therefore the fact it is impossible to do this is unimportant. In a Linda-like system values may be converted in either direction between two types so the algorithm needs to be defined to use AC-equality not AC-subtyping. Modifying the Di Cosmo algorithm to exclude subtypes from the match simply involves modifying two rules and removing one, and it is the modified algorithm that will be described below.

The algorithm is divided into two stages, the first explores the two types and creates a set, \(U\), which contains every pair of types which must have their equivalence determined in order to decide the equivalence of \(p_0\) (where \(p_0\) is the pair of types being tested). The second stage takes the set \(U\) and determines the greatest subset such that all elements contain two AC-equal types. The equivalence test for \(p_0\) then becomes a test on whether \(p_0\) is in this subset.

Stage one of the algorithm is computed as follows:

1. Let \(U = \emptyset\) and \(W = \{p_0\}\).

2. While \(W\) is nonempty do:

   (a) Take a pair \(p\) out of \(W\);
   
   (b) If \(p \in U\) continue at 2;
   
   (c) Insert \(p\) into \(U\);
   
   (d) If \(p\) is of the form \((\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2)\), then insert \((\tau'_1, \tau_1)\) and \((\tau_2, \tau'_2)\) into \(W\);
   
   (e) If \(p\) is of the form \((\Pi_{i=1}^n \tau_i, \Pi_{j=1}^n \tau'_j)\), then insert every \((\tau_i, \tau'_j)\), for \(i \in 1, \ldots, n\) and \(j \in 1, \ldots, n\), into \(W\).

\(^2\)Where AC stands for Associative-Commutative
There are two points of this algorithm which may cause confusion. Firstly rule 2 d) inserts two pairs into \( W \) but the types are in different orders. The pair containing the parameter types compares \( \tau'_1 \) against \( \tau_1 \) while the pair containing the return type compares \( \tau_2 \) against \( \tau'_2 \). This is due to covariance/contra-variance of function types. Parameters are converted from their original type into the type used by the function while the return values are converted from the type representation used by the function into the type representation used by the caller. As was discussed above this algorithm was designed to allow types of different sizes to match, and therefore this ordering is important. Secondly Di Cosmo uses \( \Pi^n \) to represent an \( n \) element product.

Rule 2 e) is the rule that was modified to force matching up to AC-equality rather than matching up to AC-subtyping. The original version allowed two different sized products to be matched whereas in the new algorithm they must be the same size. The original line was ‘If \( p \) is of the form \( (\Pi^m_{i=1} \tau_i, \Pi^n_{j=1} \tau'_j) \), then insert every \( (\tau_i, \tau'_j) \), for \( i \in \{1, \ldots, n\} \) and \( j \in \{1, \ldots, m\} \), into \( W \).’

The Di Cosmo paper does not mention sum types so they do not appear in the algorithm.

The second stage, shown below, takes the set \( U \) computed previously and maintains a three set partition \( W, S \) and \( F \). Set \( W \) is the work list, which contains the pairs for which equality has yet to be determined. \( S \) contains ‘suspended’ pairs, which are valid providing all their descendants in \( W \) are also found to be valid. The final set, \( F \), contains all the known invalid pairs.

The algorithm references an auxiliary procedure, \( \text{invalidate} \ p \). This procedure takes a pair \( p \), and moves it into \( F \). It also takes every pair which references a type in \( p \) that is currently in \( S \) and moves it back into \( W \) for further examination.

1. Let \( W = U \) and \( S = F = \emptyset \).

2. While \( W \) is nonempty:

   (a) Take a pair \( p \) out of \( W \);
   (b) If \( p \) is of the form \( (\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2) \), then if \( (\tau'_1, \tau_1) \not\in F \) and \( (\tau_2, \tau'_2) \not\in F \) then insert \( p \) into \( S \) else invalidate \( p \).
   (c) If \( p \) is of the form \( (\Pi^m_{i=1} \tau_i, \Pi^n_{j=1} \tau'_j) \), then if there exists \( \sigma \in \Sigma^m_1 \) such that for all \( i \in \{1, \ldots, n\} \), \( (\tau_{\sigma(i)}, \tau'_i) \not\in F \) holds, insert \( p \) into \( S \) else invalidate \( p \).
   (d) If \( p \) satisfies none of the three previous tests, then invalidate \( p \).

3. If \( p_0 \not\in F \), return \( \text{true} \), otherwise return \( \text{false} \).

The rule 2 d) is the rule that has been changed, the original line was ‘If \( p \) is of the form \( (\Pi^m_{i=1} \tau_i, \Pi^n_{j=1} \tau'_j) \), then if there exists \( \sigma \in \Sigma^m_n \) such that for all \( i \in \{1, \ldots, m\} \), \( (\tau_{\sigma(j)}, \tau'_j) \not\in F \) holds, insert \( p \) into \( S \) else invalidate \( p \).’ The line that was
removed was between 2 a) and 2 b) and read ‘If \( p \) is of the form \((\bot, \tau')\) or \((\tau, \top)\), then insert \( p \) into \( S \).’

This algorithm is very flexible and has the bonus that the method for calculating it is spelt out in the paper in a simple step-by-step fashion. The algorithm treats products as commutative, it handles function types in similar fashion to the structural algorithm given earlier but it does not mention sum types. Sum types can be added to the algorithm by adding an extra condition to each of the two stages. The next algorithm to be described does handle sum types and so the problem of adding sum types will be discussed in the comparison of the algorithms.

### 4.2.3 Atanassow 04

In their 2004 paper [117] Frank Atanassow and Johan Jeuring describe an algorithm for converting between data stored in an XML format and Generic Haskell [118]. Their case study uses an XML data binding called UUXML [119]. An XML data binding uses an XML schema to produce a converter that takes an XML document and gives a value in the native types of a programming language. The translation program tends to produce “overwhelmingly” complex data type definitions that are awkward and unwieldy for a programmer to use. In their extension to UUXML the programmer specifies a type and the program determines whether the data type generated by the XML data binding is isomorphic to it, and applies the inferred coercion if so. This allows the programmer to use their own representation, regardless of how convoluted the generated type structure was.

This algorithm does not explicitly regard an isomorphism as a pair of functions. Each application of the algorithm is purely a one-way operation, converting type \( A \) into type \( B \). The rules are such that should \( A \rightarrow B \) succeed then an application of the algorithm to the same two types in the reverse order will also succeed, generating the other half of the pair.

Their notion of an isomorphism is divided into two distinct parts. Firstly a value is ‘reduced’ to an intermediate ‘universal’ form, then secondly the ‘universal’ form is ‘expanded’ into the correct type. This can be summarised as \( \text{reduce}|t| :: t \rightarrow \text{Univ} \) and \( \text{expand}|t'| :: \text{Univ} \rightarrow t \). This means an isomorphism that converts from \( t \) to \( t' \) is defined as \( \text{expand}|t'| \circ \text{reduce}|t| \).

If we consider a small set of base types, namely integers, then the code given below will convert a native Haskell integer into a ‘Universal’ integer which is used in the intermediate step of the conversion process.

---

3Generic Haskell is an extension to Haskell which adds support for type indexed values and kind indexed types through the use of the special brackets \( \{ k | t \} \) and \( \{ l | k \} \).
type ReduceBase{[*]} t = UBase t
reducebase{|t :: k|} :: ReduceBase{|k|} t
reducebase{|Int|} i = UInt i

type ExpandBase{[*]} t = UBase t
expandbase{|t :: k|} :: ExpandBase{|k|} t
expandbase{|Int|} (UInt i) = i

Built on top of the ReduceBase function are the functions Red and Reduce. Red takes a value \( t \) and a accumulation list of Univ values. It returns the accumulation list with the normal form of \( t \) appended.

type Red{[*]} t = t -> Univ -> Univ
red{|t :: k|} :: Red{|k|} t
red{|Int|} i = reducebase{|Int|} i : u
red{|Unit|} () = id
red{|a :*: b|} (a :*: b) = red{|a|} a red{|b|} b

The Reduce function just primes Red with an empty list.

reduce{|t :: *|} :: t -> Univ
reduce{|t|} x = red{|t|} x []

The expand function works by removing items from the list of universal values and building up the ‘expanded’ value.

type Expand{[*]} t = Univ -> t
expand {|t :: |} :: Expand{|[]|} t
expand {|t|} u = case exp{|t|} u of
  (v, []) -> v
  (v, _) -> error "expand"

expand in turn depends on the function exp which does most of the expansion work. The return value of exp is a pair containing the expanded value and list of the values that remain to be expanded.

type Exp{[*]} t = Univ -> (t, Univ)
exp{|t :: k|} :: Exp{|k|} t
exp{|Int|} (u : us) = (expandbase{|Int|} u, us)
exp{|Int|} [] = error "exp"
exp{|Unit|} us = (Unit, us)
exp{|a :*: b|} us = let (u, us’) = exp{|a|} us
  (v, us’’) = exp{|b|} us’’
in (u :*: v, us’’)
reduce{|((Int, (Int, Int)), ()))|} ((2, (3, 4)), ()
    = [UInt 2, UInt 3, UInt 4]

The corresponding example of \texttt{expand}, which converts from the ‘universal’ type into the type given by the programmer is shown below.

\begin{verbatim}
expand{|((Int, (Int, Int)), ()))|} [UInt 2, UInt 3, UInt 4]
    = ((2, (3, 4)), ())
\end{verbatim}

As is clear from these two code snippets the Atanassow algorithm treats tuples as associative. The algorithm flattens any tuples and then builds any tuples from the destination type using the flattened values. As well as products the paper discusses treating sum types as equivalent. In essence sum types go through a similar ‘flattening’ process. Two items are stored for each sum type: the position of the type in the sum type, and the number of choices (the arity of the type). To expand a sum type the arity of the destination type is checked against the source arity. If they are not the same then the two types cannot match.

The functions \texttt{reduce} and \texttt{expand} are defined separately for product and sum types, and in the final part the \texttt{Univ} types are renamed to \texttt{UnivP} and \texttt{UnivS} to represent product and sum types respectively. A new top level \texttt{Univ} type is created which combines the two new types.

The final part of their algorithm is to model the subtyping relation that is found in XML schema and other programming languages. For example \texttt{int} is a subtype of \texttt{integer} which is a subtype of \texttt{decimal}. Support for this is added by modifying the \texttt{expandbase} function to allow more types to be converted. For example the following three type definitions will allow an \texttt{int} and an \texttt{integer} to be coerced into a \texttt{decimal}.

\begin{verbatim}
expandbase{|Decimal|} (UDecimal x) = x
expandbase{|Decimal|} (UInteger x) = integer2dec x
expandbase{|Decimal|} (UInt x) = int2dec x
\end{verbatim}

\section*{4.3 Comparing Type Isomorphism Algorithms}

When comparing type isomorphism algorithms there are two major aspects to be considered. Firstly how the types that are accepted by the algorithms differ and secondly the efficiency of the matching process. The algorithm that is accepted must conform to the requirements of running as part of a \texttt{LINDA} system, and it is also important that it is efficient as the algorithm will be run for each pair of types in a tuplespace.

\subsection*{4.3.1 Matched Types}

When the Atanassow algorithm is compared to Di Cosmo’s there are several fundamental differences in the types that are declared equivalent. The Atanassow algorithm does not consider products to be commutative, they discuss the problem of
isomorphisms that are generated but fail to convert all possible values correctly (as in different representations of atomic types), and the algorithm handles sum types. Di Cosmo on the other hand does treat products as commutative, which by itself matches a large class of types not matched by Atanassow. The lack of function types in Atanassow is a minor problem as they would not be difficult to add. Support for sum types in Di Cosmo’s algorithm is more of a concern as the algorithm is more complex than Atanassow’s.

The method of calculating equivalence is also very different. Di Cosmo builds up a set containing the isomorphisms for the two type representations and every other representation that is referenced by them. Conversely the Atanassow algorithm relies heavily on the language in which it is implemented to do much of the work automatically. Indeed it is difficult to see where the isomorphism comes from in Atanassow’s algorithm as at no point does the algorithm confirm that both parts of the isomorphism can be generated nor does it produce a function that can deal with a value, the algorithm converts the value as it goes. The fact that Atanassow’s algorithm produces a type isomorphism is due to the rules that govern the translation being chosen to guarantee any conversion is reversible. Although this hides the isomorphisms, values of any two types that can be converted \( A \to B \) can also be converted \( B \to A \).

Product types are treated differently by these two algorithms. With Di Cosmo \( A \times B \) is equivalent to \( B \times A \) while with Atanassow’s this is not the case. Allowing commutative products makes an already existing problem even more serious for any algorithm that uses them; the problem of the semantics of a type with respect to type matching has not been discussed so far. This issue will be discussed in section 4.8. Commutative products allow a much wide range of types to be declared as equivalent.

### 4.3.2 Algorithmic Complexity

One of the advantages of the Di Cosmo paper is that it features a discussion on the complexity of the algorithm. The worst case bound of the algorithm is \( O(n^2n'^2d^2) \) where \( n \) and \( n' \) count the sub-terms in the two types and \( d \) is a bound on the arity of the products involved. The authors state that “the complexity of the algorithm may also be bounded by \( O(NN'd'^2) \), where \( N \) and \( N' \) are the sizes of the types being compared. In practice, \( N \) and \( N' \) might be significantly less than \( n^2 \) and \( n'^2 \), respectively.”

The structural equivalence algorithm is a simple algorithm and the complexity is \( O(nd) \) where \( n \) is the number of types in the type being compared and \( d \) is a bound on the arity of the products being compared. If the types are of different sizes then the algorithm will fail before any of the constituent types are compared.

Atanassow’s paper does not discuss the computational complexity of the algorithm. As the algorithm involves two passes over the value — one to reduce the value, the other to expand — this gives the algorithm a complexity of \( O(nd + n'd') \).
Ordering the three algorithms gives the structural equivalence algorithm the lowest time complexity, Atanassow is marginally worse and Di Cosmo’s worst case places it last. The complexity of the three algorithms exactly follows the size of the set of types that are declared isomorphic by the algorithms. The structural equivalence algorithm accepts less than the requirements specified, the lack of reordering of sum types is very restrictive and needs to be relaxed. Atanassow’s algorithm also does not accept reordered sum types.

Di Cosmo’s algorithm actually accepts far more types than was given in the requirements at the beginning of the chapter. Reordering of values in products is too expensive and too relaxed for use in a system that must match types whose values will be converted in both directions, in real time. The reason Di Cosmo’s algorithm was included is because it uses an interesting, well designed algorithm for generating the isomorphisms that has clearly been written with implementations in mind. This is something many papers on the subject lack, and the implementation is a vital part of the work presented in this thesis.

4.4 Decision

The three algorithms presented here are just a small selection of papers on the subject, but these algorithms show the diversity in the topic. They all declare different classes of types to be isomorphic, perform the calculations in different ways and have different complexities. None of them matches the requirements laid out at the start of this chapter, for example none of the algorithms deal with pointers. The Atanassow algorithm comes closest, with only support for functions and pointers missing. Atanassow’s algorithm also has a very low complexity. The Di Cosmo algorithm, although it has a higher complexity, has the advantage that as a side effect it calculates all isomorphisms between a group of types rather than working solely on pairs. The sets of generated isomorphisms can be stored after the initial test has been performed and then a future type can be added to the set. When the algorithm run again and the isomorphisms will be calculated between the new type and the types previously in the set, without recalculating the isomorphisms that were generated previously.

As will be discussed in later chapters caching of type isomorphisms is a vital component of the work proposed in this thesis. Garbage collection of the generated isomorphisms is also required and would have to be added to the system. Rather than complicate the algorithms even further the type isomorphisms will be generated independently and the caching and garbage collection performed using a different algorithm.

The extensions to the Atanassow algorithm that need to be implemented are to support functions and pointers, and to make the order of types in sum types unimportant. Functions are a straightforward addition to the reduce-expand pair of functions. A function type ‘reduces’ to a UFunction type, where the two arguments are reductions of the two arguments to the original function type. The UFunction
type is then ‘expanded’ into a function type where the two arguments are ‘expansions’ of the two arguments to \texttt{UFunction}. Pointers are unaffected by reduction and expansion. They are only matched if the two types pointed to by the pointers are isomorphic.

The largest change required is to make sum types reorderable. The new rules reduce a sum type to a set of ‘reduced’ types. This set is then ‘expanded’ into the list of types in the type it is being converted into. To retrieve the function the order in which the types were rearranged is stored, and can then be used to retrieve the correct \texttt{expand/reduce} functions.

The Atanassow is efficient and allows a wide range of type representations to be declared isomorphic. It is the algorithm that will be used in the implementation of the proposed system.

4.5 Algorithm

When introducing the Atanassow algorithm the full algorithm was not given as it is too long and too heavily integrated with Generic Haskell to be easily separated. Below is the implementation of the algorithm used in the rest of the thesis. It has been simplified slightly for clarity, the main simplifications have been in the removal of code to check if two types have already been converted and the removal of code which allows the generated isomorphism to be transferred to client programs (as used in the conversion of function values). The code for pointers has also been simplified to remove code for handling cyclical data structures. These simplifications help with the overall understanding of the algorithm and do not affect the time complexity of the system.

The function \texttt{compare\_types} takes two type objects \texttt{t1} and \texttt{t2} and returns a function which accepts a value of type \texttt{t1} and returns one of type \texttt{t2} if the types are equivalent, and \texttt{None} if not. The function is essentially a large case statement divided up based on the class of type that is passed in. If the type is a type name which is a built in then the function is the identity isomorphism. If it is not a built in type then the actual types are looked up and compared.

If the type is a product the function \texttt{flattenProductType} is called on both types. This returns a flat list of the elements of each product, so \((a, (b, (c, d), e))\) becomes \((a, b, c, d, e)\). This is equivalent to the \texttt{Reduce} function in the original paper. The two flattened products are then compared. The isomorphism function that is returned flattens the value, converts each member of the product in turn and then calls \texttt{raiseProductValue} which performs the opposite of \texttt{flattenProductType}, but it works on a value.

The sum type case works in a similar flattening/expanding fashion as products did. Rather than perform a straight comparison between the two flattened types all possible permutations are found of the mapping between elements in the two flattened types. Each permutation is tested in turn until a valid mapping is found. In testing each permutation most pairs of types will be compared more than once.
The actual implementation includes code to only compare each pair once and to cache the result.

The function and pointer cases are both simple and require little explanation. The pointer case has been simplified and in the actual implementation features code to handle cyclic values by using a memo structure.

```python
def compare_types(t1, t2):
    if t1.isId() and t1.id not in builtin:
        return compare_types(lookupType(t1.id), t2)
    elif t2.isId() and t2.id not in builtin:
        return compare_types(t1, lookupType(t2.id))
    elif t1.isId() and t2.isId():
        if t1.id == t2.id:
            return lambda x: x
        elif t1.isProductType() and t2.isProductType():
            t1e = flattenProductType(t1)
            t2e = flattenProductType(t2)
            if len(t1e) != len(t2e):
                return None
            funcs = []
            for e1, e2 in zip(t1e, t2e):
                f = compare_types(e1, e2, checked)
                if f is None:
                    return None
                else:
                    funcs.append(f)
            return lambda x: raiseProductValue(t2,
                                                map(funcs, flattenProductValue(x)))
        elif t1.isSumType() and t2.isSumType():
            t1e = flattenSumType(t1)
            t2e = flattenSumType(t2)
            if len(t1e) != len(t2e):
                return None
            for perm in all_perms(range(len(t2))):
                funcs = []
                for i in range(len(perm)):
                    f = compare_types(t1e[i][0], t2e[perm[i]][0])
                    if f is None:
                        break
                    else:
                        funcs.append((f, perm[i]))
                if len(funcs) == len(perm):
                    break
        return lambda x: x
```
if len(funcs) != len(t1e):
    return None
else:
    return lambda x: raiseSumType(t2, funcs[x.index][1],
                                   funcs[x.index][0](x))

elif t1.isFunctionType() and t2.isFunctionType():
    arg = compare_types(t1.arg, t2.arg)
    returnval = compare_types(t2._return, t1._return)
    if arg and returnval:
        return lambda x: returnval(x(arg))

elif t1.isPointerType() and t2.isPointerType():
    f = compare_types(t1.ptr, t2.ptr)
    if f:
        return lambda x: ptr(f(x.value))

Let us consider three type specifications, A :: int32 + (int32 * ptr(A)), B :: (int32 * C) + int32 and C :: ptr(B). The specifications B and C form a pair, and together they are isomorphic with A. The algorithm given above determines this as follows. The flattening and expanding of product and sum types have been excluded from the description as they have no effect on this examples. The following example will show these rules in action.

1. Compare A and B. Type A is an id, but not a built in id so look the type up and compare.

2. Compare int32 + (int32 * ptr(A)) and B. Type B is an id, but not a built in id so look the type up and compare.

3. Compare int32 + (int32 * ptr(A)) and (int32 * C) + int32. Both are sum types they have the same arity. The first possible permutation is found and compared.

   (a) Compare int32 and int32 * C. int32 is a built in id while int32 * C is a product type so they are not isomorphic.

4. Previous permutation failed, try the next one.

   (a) Compare int32 and int32. Both types are built in ids and are identical so they are isomorphic.

   (b) Compare int32 * ptr(A) and int32 * C.

      i. Compare int32 and int32. Both types are built in ids and are identical so they are isomorphic.

      ii. Compare ptr(A) and C. C is an id, but not a built in id so look the type up and compare.
iii. Compare \( \text{ptr}(A) \) and \( \text{ptr}(B) \). Types \( A \) and \( B \) are already being compared so assume they are isomorphic.

(c) Types \( \text{int32} \ast \text{ptr}(A) \) and \( \text{int32} \ast \text{C} \) are isomorphic.

5. With the mapping \( A_0 \rightarrow B_1, A_1 \rightarrow B_0 \) types \( \text{int32} + (\text{int32} \ast \text{ptr}(A)) \) and \( (\text{int32} \ast \text{C}) + \text{int32} \) are isomorphic.

6. Types \( A \) and \( B \) are isomorphic.

To complete the isomorphism we will compare the types \( A \) and \( B \) in the reverse direction.

1. Compare \( B \) and \( A \). Type \( B \) is an id, but not a built in id so look the type up and compare.

2. Compare \( (\text{int32} \ast \text{C}) + \text{int32} \) and \( A \). Type \( A \) is an id, but not a built in id so look the type up and compare.

3. Compare \( (\text{int32} \ast \text{C}) + \text{int32} \) and \( \text{int32} + (\text{int32} \ast \text{ptr}(A)) \). Both are sum types they have the same arity. The first possible permutation is found and compared.

   (a) Compare \( \text{int32} \ast \text{C} \) and \( \text{int32} \). \( \text{int32} \ast \text{C} \) is a product type while \( \text{int32} \) is a built in id so they are not isomorphic.

4. Previous permutation failed, try the next one.

   (a) Compare \( \text{int32} \ast \text{C} \) and \( \text{int32} \ast \text{ptr}(A) \).

      i. Compare \( \text{int32} \) and \( \text{int32} \). Both types are built in ids and are identical so they are isomorphic.

      ii. Compare \( \text{C} \) and \( \text{ptr}(A) \). \( \text{C} \) is an id, but not a built in id so look the type up and compare.

      iii. Compare \( \text{ptr}(B) \) and \( \text{ptr}(A) \). Types \( B \) and \( A \) are already being compared so assume they are isomorphic.

   (b) Compare \( \text{int32} \) and \( \text{int32} \). Both types are built in ids and are identical so they are isomorphic.

   (c) Types \( \text{int32} \ast \text{ptr}(A) \) and \( \text{int32} \ast \text{C} \) are isomorphic.

5. With the mapping \( B_0 \rightarrow A_1, B_1 \rightarrow A_0 \) types \( (\text{int32} \ast \text{C}) + \text{int32} \) and \( \text{int32} + (\text{int32} \ast \text{ptr}(A)) \) are isomorphic.

6. Types \( B \) and \( A \) are isomorphic.

The above shows that the two types tested are equivalent when compared in both directions, and thus are isomorphic. If \( A \) were compared with type \( C \) the first step would determine that \( A \) is a sum type while \( C \) is a pointer and thus they are not isomorphic.
Next we shall consider the subtly different types $D :: \text{int32} \times \text{int32}$ and $E :: \text{int32} \times (\text{int32} \times 1)$. The $\text{(int32} \times 1)$ in the type $E$ indicate that this is a one element product type. Just as in C the following structure is different from a plain integer, so $(\text{int32} \times 1)$ is different from $\text{int32}$.

```c
struct x {
    int y;
};
```

Although different $D$ and $E$ are isomorphic, and this is verified by the algorithm.

1. Compare $D$ and $E$. Type $D$ is an id, but not a built in id so look the type up and compare.

2. Compare $\text{int32} \times \text{int32}$ and $E$. Type $E$ is an id, but not a built in id so look the type up and compare.

3. Compare $\text{int32} \times \text{int32}$ and $\text{int32} \times (\text{int32} \times 1)$. Flatten both product types and compare.

4. Compare $\text{int32} \times \text{int32}$ and $\text{int32} \times \text{int32}$. Both types are product types and have the same arity.
   (a) Compare $\text{int32}$ and $\text{int32}$. Both types are built in identifiers, and are identical, so they are isomorphic.
   (b) Compare $\text{int32}$ and $\text{int32}$. Both types are built in identifiers, and are identical, so they are isomorphic.

5. Type $\text{int32} \times \text{int32}$ and $\text{int32} \times (\text{int32} \times 1)$ are isomorphic.

6. Types $D$ and $E$ are isomorphic.

To confirm the isomorphism the algorithm is run over the opposite pairing.

1. Compare $E$ and $D$. Type $E$ is an id, but not a built in id so look the type up and compare.

2. Compare $\text{int32} \times (\text{int32} \times 1)$ and $D$. Type $D$ is an id, but not a built in id so look the type up and compare.

3. Compare $\text{int32} \times (\text{int32} \times 1)$ and $\text{int32} \times \text{int32}$. Flatten both product types and compare.

4. Compare $\text{int32} \times \text{int32}$ and $\text{int32} \times \text{int32}$. Both types are product types and have the same arity.
   (a) Compare $\text{int32}$ and $\text{int32}$. Both types are built in identifiers, and are identical, so they are isomorphic.
(b) Compare \texttt{int32} and \texttt{int32}. Both types are built in identifiers, and are identical, so they are isomorphic.

5. Type \texttt{int32 * (int32 * 1)} and \texttt{int32 * int32} are isomorphic.

6. Types \texttt{E} and \texttt{D} are isomorphic.

Again, and as expected, these two types are determined to be isomorphic.

4.6 Pointers

Pointers are a unique and interesting class of types because they contain a single value of a specific type, but that value can be shared between other pointer values. This means it is not possible to simply recurse over the type when comparing it, nor is it possible to recurse over the value while converting it. Both these cases risk an infinite loop. When comparing a type a memo structure is used which tracks which pairs of types have already been compared, or are in the process of being compared. Before each pair of types are compared the system checks whether they exist in the memo structure and if so the stored result is returned. If they are in the process of being compared then they are assumed to match. If they do not match it will be found as part of the rest of the matching process.

The value of a pointer is the address in memory of the value it points to. When serialising a value it is necessary to convert this address into an identifier which can be used by the deserialising process to find the correct value in the serialised stream. This is also a problem when converting the value of a pointer as pointers can create loops or more than one pointer can point at the same value. To prevent the same value being converted more than once and to prevent the fact that the two pointers point at the same value being lost a memo structure is used which tracks which values have been processed and to which values they have been converted. When a value is processed an empty structure is created which will hold its converted value. The address of this value is added to the memo structure and the conversion of the value begins. It is necessary to add an empty structure to the memo field so that if any pointers create a cycle an infinite loop is not entered.

4.7 Retrieving Conversion Functions

Calculating a match between two types using a type isomorphism algorithm is not just a yes/no question. To indicate that two types match, the algorithm must return a function that can be stored and used by the kernel to convert values between the two types. The algorithm need not return both parts of the isomorphism, as long as returning one guarantees that the inverse function will be successfully generated. For an algorithm \( A \) and two types \( t_1 \) and \( t_2 \) if \( A(t_1, t_2) = f \) (and \( A(t_1, t_2) \neq \perp \)) then \( A(t_2, t_1) = g \) (where \( f \) and \( g \) comprise an isomorphism) must hold.
The benefit of returning both functions at once is that by being generated simultaneously they may share some of the work which would otherwise be duplicated. If the algorithm is designed to be run twice then it will end up being simpler and the reverse isomorphism may not be needed because of the way the client processes interact with the system. Calculating as few isomorphisms as possible is going to be a key feature of the optimisation of the system proposed in this thesis therefore the algorithm used will only generate one function at a time, but by successfully generating one half of the isomorphism it is guaranteed that the reverse will be generated too.

Whether the algorithm calculates the isomorphism in one or both directions there still needs to be a method for the algorithm to return a recipe for converting values between types. The theory states that this is a function, but implementing it may not involve functions as they exist in the language. Python, the language used in the implementation of the proposed system, does allow functions to be created dynamically at runtime but other languages that could be used, such as C, do not. In a language such as Python the simplest way to build a function at runtime is to construct it out of many sub-functions but for languages that do not allow this the algorithm can produce a sequence of instructions which can be evaluated by an interpreter. Correctly implementing this will prove to be fast, although obviously not as fast as if the functions were implemented as native functions (due to the overhead of interpreting the instructions). For the sake of simplicity the implementation of the work uses Python functions to build up the isomorphisms.

4.8 The Semantic Problem

So far types have been discussed as if their values have no meaning. Values are not simply defined by their type, the programmer implicitly gives the value meaning perhaps through the name of the type, or just through the way in which it is used. A classic example of this problem is degrees and radians. Both of these types measure angle, and both can naturally be represented as floating point numbers yet they are not the same type. It is true that they are isomorphic, but you cannot use a value from one where you would use a value of the other type without a conversion. This problem is most readily apparent in the case of OpenGL [120] whose functions that take angles as an argument expect degrees, yet in C the standard trigonometric functions take and return radians. In this case the compiler will not pick up the error, even though the values are of different (semantic) types.

The semantic problem can also be seen if products are treated as commutative. Take a pair of Cartesian coordinates, they are naturally represented as a pair of floating point numbers yet which element of the pair is which element of the coordinates? It is natural to assume that the first element will be the ‘x’ coordinate and the second to be the ‘y’ but there is nothing in the type specification to guarantee that this is the case.

In a traditional LINDA system a Cartesian coordinate would be represented as a
two element tuple with both elements as floating point numbers. A Polar coordinate would also be represented as a two element tuple with both elements as floating point numbers. It is impossible for a process trying to match a Polar coordinate or a Cartesian coordinate to exclude the other semantic meaning from the returned tuples. Although in the modified system this problem still exists, a process will not accidentally match a two element tuple where the elements are independent values as the coordinates will be represented as a single element tuple with a pair as the first element.

Neither of the two papers which introduced the algorithms above made any mention of the ‘meaning’ of the values. In the original LINDA where the only types allowed were simple atomic data types the same problem existed. Values in LINDA have always been matched purely based on their type signature or their value, and it is not the intention of this work to change this. Only the method by which the types are matched is being altered, not the idea behind it.

4.9 Conclusion

In this chapter the historical use of type isomorphisms has been discussed along with an outline of their proposed use in this thesis. By using type isomorphisms to determine the equivalence of two types it has been shown that a range of types can be used as if they were identical. It has also been shown that all that is needed to allow types to be used in this way is a type specification that represents the values of the types. The various algorithms that were discussed all take two type specifications and return a pair of functions to convert values between the two types. These functions can then be applied to values which are then converted to the equivalent value in the opposite type.

A brief description of the algorithmic complexity of the described algorithms was given. The ‘real-world’ performance of the modified version of the Atanassow algorithm will be discussed in chapter 8. Since type matching is such a vital part of a LINDA-like system and the type matching algorithm is called so frequently it is important that the algorithm is fast enough for an average type that the system does not become so slow as to be unusable.

Two elements of the compound type matching puzzle are now in place. Types can be described in any language and transmitted to a LINDA kernel and the kernel can determine if two types are isomorphic and convert values between the two. In the next chapter it will be shown how these can be integrated with a LINDA-like system while maintaining its LINDA-like properties.
Chapter 5

Functions

Up to this point functions have been treated the same as the other types that have been considered. Unfortunately, implementing functions in the proposed system requires special consideration if it is to satisfy the goals of the project, in particular to allow cross-language, cross-platform communication while maintaining the open and distributed nature of Linda. In this chapter the reasons behind giving functions special treatment will be considered, the various options available to solve the problems presented by functions will be outlined and finally one of them will be chosen.

Functions are lumps of code that accept zero, one or more values and return a single value as the result of executing the code. It is important to make the distinction between functions and processes, at least from a logical perspective. A function is simply a single piece of code which in most cases cannot be executed on its own as it depends on other functions. A process is made up of one or more functions, with all their dependencies fulfilled.

For functions to fit naturally into a open coordination system a process should be defined using functions as normal. When creating a tuple it should be possible to include a function as one of the values, or as part of one of the values. The function should be placed into a tuplespace so that it can be retrieved and used by a process written in a different language, possibly running on a different machine. This process should be able to use the retrieved function as though it were native to it, no matter what the differences in language or hardware architecture are.

As an example of the ideal figure 5.1 shows a process written in C takes a function, double and places it into a tuple. This tuple is then retrieved and the function evaluated in Python.

Figure 5.2 shows how the function double could be added to a tuple and placed in to the uts.

The Python code above creates a type specification which will match the type of the function from the C code, performs an in on the uts and then calls the function. The result of running these two programs should be that the Python process prints ‘4’ to the standard output. In this chapter the reasons why it is not practical to implement a system that works exactly like this will be discussed, as
int double(int x) {
    return x * 2;
}

int main(int argc, char* argv[]) {
    LindaValue tup;
    Linda_connect();
    tup = Linda_tuple(1);
    Linda_setTuple(tup, 0, double);
    Linda_out(Linda_uts, tup);
}

Figure 5.1: Idealised C code for communicating with LINDA.

import linda
linda.connect()

type = linda.Type("f :: int -> int;")

tup = linda.uts._in((type, ))
func = tup[0]
print func(2)

Figure 5.2: Idealised C code for communicating with LINDA.

well as proposing a system that works in a similar fashion and can be implemented relatively easily.

5.1 The Problem With Functions

The aim of the work presented in this thesis is to allow a value of any type, from any programming language, to be placed inside a tuple which is used to communicate through a LINDA-like system. The type of this value is matched against isomorphic types and then matched values can be converted into the equivalent value for the isomorphic type. This is all done transparently inside the kernel and allows processes to retrieve values without caring what process, programming language or hardware architecture they came from.

The type matching algorithm treats types as though they are one of five kinds: atomic types, products, sums, pointers and functions. Converting between values of the types in any of the first four classes is relatively straight-forward. There is a limited number of atomic types, with well defined differences between the various representations. Products are just a simple wrapper around other values, the same
is true of sum types. Values of function types do not contain a value that is of any of the other four kinds, instead they contain the code for the function.

The code that represents a function is highly dependent on the language used and hardware architecture that it is compiled for. Optimisations performed by a compiler may remove a function from the compiled code entirely, placing it inline wherever it is used. The output produced by a compiler for a function depends heavily on the hardware architecture that it is being compiled for. A C compiler will produce machine code optimised for a specific type of processor, but also the code maybe altered depending on the flags passed to the compiler and the calling conventions used. Just as different processor architectures have different machine codes, the virtual machine code that is used is different for every virtual machine and is in turn different from any actual hardware’s machine code. Python specifies that the virtual machine is not standard and while the language itself may be backwards compatible between versions no such guarantee is made about the virtual machine, which may be subject to complete change between even minor releases.

Some languages, such as C, do not treat functions as first class objects. C does allow function objects to be ‘faked’ by treating the name of the function as a pointer which can be passed as a value. In keeping with the rest of C’s type system nothing can be queried about the function from the pointer. The pointer is the address of the first byte of the compiled function’s code. There is no direct way to find out the size of the function’s compiled code, other than by tracing possible execution paths.

Languages that do treat functions as first class values, such as Python or Haskell, allow functions to be passed around in exactly the same manner as other objects. While in most cases when programming a single process that involves no coordination this makes little difference compared to using function pointers. However, when coordination is involved, and functions are to be transferred as values, then treating functions as first class values makes much more sense.

5.1.1 Input/Output Operations

Input/output operations are used by programs to interact with the world outside their process. Whether it involves accessing the network, reading information from a file or printing information onto the screen the key component is that the information is being transmitted to or received from an outside source. The source or destination of this information is specified by a name, either a network URL, a path on the file system or a built-in stream such as ‘stdout’.

Functions that access outside sources are no longer reliant purely on the current state of the program, instead they are also dependent on the physical state and position of the machine. The range of accessible network addresses can change depending on the location of the machine the process is running on. Paths in a file system are highly machine dependent, as are standard streams such as writing to the program’s console.

If a function that uses an input/output operation is transferred to run inside a
different process then these external sources may no longer be accessible, or may
be different from the resources that were available to the original process. This
is particularly true of the program’s console. Although the original process may
have been run with a user watching its output waiting to respond to any questions,
this may not be the case for another process in the system. The process which
retrieves the function may be running in a huge server rack with no method of
communicating with the user. In this case what should the process do? If the
input/output operations act as if they are being run from the new computer then
the function will halt, waiting for a user response that will never come. If the
input/output operations behave as if they were actually run on the original machine
then the user could be overwhelmed with requests for input as the function may
have been replicated many times and be running on hundreds of machines.

The second of the two options, that input/output streams work as if they were
still being accessed on the original machine, is impossible to implement as an open
LINDA system. Such as system is designed such that processes and entire computers
can join and leave the system at will. When a value, in this case the function, is
placed into a tuplespace it will remain there even if the original process leaves the
system. Therefore it is perfectly possible for a function to still be in use long after
the original computer has left the network.

5.1.2 Global variables

Languages which allow global variables to be used pose an additional problem when
transferring functions between processes. Global variables are used to hold state
information that is accessible to the whole program. Functions can access and set
global variables to alter the state of the program, and that information will persist
across multiple calls to the function and calls to other functions.

When a function is transferred to another process there are two options for
handling any global variables that are referenced by the function. A closure must be
taken, which join all referenced variables and functions into a single object. Either
a copy of this closure can be taken and transferred with the function, or accesses
to the variables can be redirected back to the original process and their values used
from there. The second of these two options suffers from the same problems as the
input/output operations, namely that the original process can not be guaranteed
to still exist. The first option has the problem that the programmer may rely on
the global variables to be accessible to every other copy of the function. If the
programmer is aware that the function will be transferred to different processes
then it is straightforward to avoid that programming technique.

5.2 Alternatives

As outlined in the previous section allowing functions to be transferred between
processes is a difficult problem. Here four solutions to the problem are outlined and
the advantages and disadvantages of each discussed.

5.2.1 Converting Code

Types other than functions need no special treatment — if they are represented differently then the values are converted between the two representations. One alternative for functions is to do the same and convert between the different representations. Each language binding can provide the LINDA kernel with a pair of functions. These functions convert to and from the compiled code and a specified intermediate language. With these the kernel can convert from any compiled language to any other compiled language.

Were functions converted between the different representations to be used, then writing a language binding would become significantly harder than without them. Machine code is designed to be easy to execute, not easy to parse and convert into a different format. With enough resources and time, writing the conversion functions for every architecture and virtual machine would be possible. The level of difficulty is so high that it is impossible to achieve in practice for the range of languages and hardware that the proposed system aims to support.

Functions often call other functions, whether they be written by the programmer or included from a library. Before transmitting a function the language binding would have to look through the function’s code and determine which other functions might be called and transmit those as well. Standard library functions would either need to be transmitted as if they were written by the programmer, or linked to the equivalent library functions when the program was received by another process. Clearly this relinking could only work for a limited set of functions that are available across all platforms, but as many functions (e.g the memory allocation function malloc) rely on system calls there is no alternative but for them to be linked against the ‘native’ version.

The conversion functions would not deal with the final set of problems specified in section 5.1.2, namely global variables and input/output operations. In cases where the programming language is compiled to machine code, the names of variables are often lost and are simply addressed by a number in the code. Calculating where the machine code refers to a variable and whether this variable is global or local is a complicated task. Variables may also be optimised away by the compiler, or introduced when there was no variable in the source code making it impossible to determine what should be transmitted with the function value.

Allowing any language to have functions transferred across the network would impose no limits on the use of input/output operations from functions. One solution is to require the programmer not to use any input/output operations. If the programmer attempts to write to a file then the operation should be undefined.
5.2.2 Interpreting Code

Rather than converting back from an intermediate code format to the receiving host’s format each language binding could implement an interpreter for the intermediate code. This would avoid the expense of performing the conversion and relinking any of the standard library functions.

An interpreter for each intermediate language could be included as part of every language binding (something that is required to communicate with the LINDA kernel). The language binding would also contain a converter which takes native code and returns the same code in the intermediate language. This would approximately halve the difficulty of writing the part of the language binding that deals with functions, as the conversion would only be in one direction. This would still require the native code to be converted to the intermediate language with all the many problems this would entail.

Interpreting the intermediate code would not handle the problems of global variables or input/output operations. However, it is possible that interpreting the code would make it easier to catch cases where these were used.

5.2.3 Static Code

To avoid the problems associated with moving functions between processes, one solution is to keep the function on the same machine and instead pass a reference that can be used in an RPC like call. The advantages of this are that any function, written in any language can be used. There are no problems with different machine codes, input/output operations or global variables. The disadvantages of this method are that when a function value is copied (through a \texttt{rd} on a tuplespace) the function is not duplicated, and all calls will come back to the original machine causing a significant bottleneck.

The most significant disadvantage to this solution, and the one that means it cannot be used, is that the function will only remain usable if the original process is still running. This cannot be guaranteed by a LINDA system so despite the advantages given above, this solution cannot be used.

5.2.4 Single Language

The final possible solution that will be considered is for functions to be written in a single language. If the language is carefully chosen all the problems with functions can be overcome, at the expense of forcing the user to write part of their program in a specified language.

The code shown in figure 5.3 contains a small piece of Haskell code embedded into a string. This code is compiled and can be used to all intents and purposes as if it were a native function. The function is then placed into a tuplespace, retrieved and used as a native function. Some languages may not allow functions to be integrated to the same extent that Python does, but as the functions are already being treated
double = linda.Function(""
    double :: int -> int
    double x = 2 * x
"")

print double(2)
linda.uts._out((double, ))

double, = linda.uts._in((linda.Type("x :: int -> int"), ))

Figure 5.3: Haskell code embedded into Python

LindaValue double = Linda_function(" \
    double :: int -> int \
    double x = 2 * x \
");

LindaValue args = Linda_tuple(1);
Linda_setTuple(args, 0, Linda_int(2));

printf("%i\n", Linda_getInt(Linda_call(double, args)));

LindaValue tup = Linda_tuple(1);
linda.uts._out(tup);

Linda_setTuple(tup, 0, Linda_type("x :: int -> int"));
r = linda.uts._in(tup);
double = Linda_getTuple(r, 0);

Figure 5.4: Haskell code embedded into C

differently from code written in the host language this is not a problem. Figure 5.4
shows the same code as figure 5.3, only converted into C code. The code is more
complex due to C’s lack of a built in tuple type. The important part of the code is
the function call Linda_call. This takes a function object and tuple containing the
arguments, and evaluates the function.

The chosen language must fulfill several specific requirements. Firstly it must be
pure, to avoid the problems with global variables. Secondly it must be possible to
prevent the user from using input/output operators. Thirdly the language must be
compilable to a platform independent virtual machine, an implementation of which
it is possible to run on all machines, and finally the language must itself have access
to LINDA and be a fully fledged LINDA client language.
5.2.5 Summary

Of the four possible implementation strategies outlined above only the single language implementation is practical to implement. Although using conversion between different machine and byte codes would provide the cleanest implementation from the programmer’s point of view, the difficulty in implementing the conversions and the problems with global variables, input/output operations and creating a closure that contains everything needed to evaluate the function, make it impractical.

The only disadvantage for the single language implementation is the lack of distinction that is made between functions which can be placed into tuples, and those that can’t. It is hoped that the case-studies shown later, and the obvious ease in implementing this method will provide a convincing argument that shows the advantages of being able to use functions in a LINDA-like system, outweigh those caused by the use of a second language.

5.3 Security

In the current environment on the Internet, spy-ware, viruses and hackers are all serious concerns for those who use it. Allowing functions to be transferred between processes would seem to make life easier for hackers and could easily be used to cause a denial of service attack. Functions are retrieved based on their type signature and not based on the meaning of the function. A function of an integer to an integer could double the parameter, triple it, add two, or any number of other possibilities. An attacker only needs to place a function with the appropriate type signature into a tuplespace and a process may retrieve it. When more than one value matches a specific template one of the values is nondeterministically chosen and returned. This has the effect that even if there is a legitimate function there is a chance that the attacker’s function will be retrieved. Once a function has been retrieved and is evaluated there is little that can be done to protect the process from any malicious intent and it is a property of the LINDA model that the programmer must protect themselves from getting the ‘wrong’ value (see section 4.8 about the semantic problem).

Of the four alternatives described previously the first was the least secure. This solution was built around a conversion between machine code and some intermediate format, and back again. If the function is transferred between two identical machines then this conversion could be skipped. By carefully crafting the machine code it would be possible for an attacker to gain complete control of a machine. Even worse is the fact that this solution did not restrict the code that was used by the function. Operating system calls such as fork would be accessible to the attacker, and can easily be used to crash or otherwise disable the machine.

The second solution involved interpreting the intermediate language rather than compiling it back to machine code. During interpretation dangerous operating system calls such as fork could be disallowed, but this would not eliminate the danger
of denial of service attacks through infinite loops. A common solution is to place a
time limit on the execution of functions. However, as will be shown later, functions
could be used to implement LINDA’s `eval` primitive which could easily be wrongly
terminated by such a mechanism.

The static code solution does not involve moving the function to the new node.
This would make it impossible for the function to perform a denial of service attack
on the machine that is calling it. Despite this, if the function never returned, the
process that made the call could be permanently blocked.

The final alternative given uses a single language which is compiled to a portable
byte code. This has the same security concerns as the second solution. The language
would disallow use of dangerous operating system calls, but would still be vulnerable
to denial of service attacks.

Although the static code solution is the most secure of the alternatives it has
already been discounted as it is impossible to implement while keeping the LINDA
principles intact. The two interpreted solutions provide the most secure environment
in that they can prevent the function from accessing any part of the system that
it should not have access to. Interpreting the function’s code does not provide
security against the function deliberately busy waiting to increase the CPU load of
the system, flooding the LINDA network with excess tuples or from giving wrong
answers to requests.

Ultimately allowing functions retrieved from unknown processes will always be
a security risk. The aim of adding compound data types, specifically functions, was
purely for the programmer’s benefit and not to enhance the security of the system.
Open implementations of LINDA have always allowed any process to connect to
them. These processes could flood the system with tuples, or interfere with the
correct operation of the system in a number of ways so the problems addressed in
this section are not new.

An extension to LINDA involving the use of capabilities [121] may help to re-
solve some of these difficulties. Capabilities can be thought of as tickets which give
processes certain privileges over certain kinds of data. Capabilities can be used to
provide a web of trust ensuring the integrity of the data in the system. If a process
accepts a function which was `outed` with a certain capability then the process can
trust that function. As capabilities involve big changes to the LINDA model they
will not be considered further, but future work could focus on how they can help
provide security against rogue functions.

### 5.4 Minimal Language

With no suitable solution available for solving the problems regarding functions
written in multiple languages a single language must be chosen. This language
must have a highly portable implementation, support the creation and transfer of
closures, be able to be closely integrated with a number of host languages and be
closely integrated with the language used to describe types. While there are few
language implementations that meet these all but the final requirement. The final, key requirement is not met by any language. If an existing language was used then not only would values have to be serialised to transfer them across the network, but they would also need to be converted into values suitable for use in the other language as well.

The creation of a new language is a big step which introduces a great deal of complexity, that ideally would be avoided. If an existing language were used then the system would benefit from using established, bug free source code and would have a much more complete language specification and standard library. This would have the disadvantage of making the system rely on a language not designed to the unusual task that this thesis would require it to fulfill. The requirement that the language be tightly integrated with the type specification language used, would require deep integration with the language implementation, something that may not be possible. Rather than force the implementation of the ideas in this thesis to work around the design choices of a third party language a new one will be created which exactly matches the requirements of the work.

In this section a new language is introduced which supports all the features required of a single-language solution, and uses the same type specification language that was described in chapter 3. The details of the language are not vital to this thesis and so a detailed description will be given in appendix B, only the practicalities of using the language and the language features with regard to mobile code are given here.

The language is a combination of the type specification language presented earlier and a simple functional language. The type system is exactly that which was presented earlier. This has the benefit of allowing the language to integrate transparently with the LINDA type matching algorithms already presented. In addition to this the language will take advantage of the type registration and conversion that has already been introduced.

5.4.1 Syntax

The syntax of Minimal is, as the name would suggest, minimal. A larger description of the language’s syntax is given in appendix B. The language borrows some of its style from Haskell, especially the specification of functions. Below is a simple function for doubling an integer value. The first line specifies the type of the function, while the second defines it.

\[
\text{double :: int -> int;}
\]
\[
\text{double } x = x * 2;
\]

The next function shown uses an if-then-else construction to return the absolute value of an integer.

\[
\text{abs :: int -> int;}
\]
\[
\text{abs } x = \text{if } x > 0 \text{ then } x \text{ else } -x \text{ endif;}
\]
The next example demonstrates how to call other functions. In this case it is a recursive call to calculate a Fibonacci number.

```haskell
fib :: int -> int;
fib x = if x == 0 then 0
  else if x == 1 then 1
  else fib(x-1) + fib(x-2)
  endif
endif
```

5.4.2 Built In Functions

The Minimal language comes with a small set of useful functions, provided to make it simpler for the programmer to perform common tasks. This include \texttt{sin, cos, abs} and other simple mathematical functions. The Minimal language also has built in support for accessing Linda. The functions \texttt{out, in, rd} and the rest are all present, except for the \texttt{connect} function as it is expected that if Linda support should be enabled in Minimal then the host program will have already enabled it.

5.4.3 Implementation

The Minimal language is implemented as a C library which is designed to be embedded into other languages. In this section the implementation of the library will be discussed and how it can be integrated with a C program. When a function, or group of functions, is created then the library is given a chunk of source code as text. This text is parsed and errors detected immediately. The parser returns a parse tree which is stored in a mapping from function names to the function’s syntax tree.

Unlike many languages the library does not compile the source code into either native code or an intermediate byte code. Most languages rely on complete source code files, but Minimal cannot as it is designed for functions to be created, at runtime, from within another program. Minimal functions can be created on a function by function basis, so the host programming language holds a reference to each of them. This is due to Minimal’s origins as a language for replacing individual functions so they can be transferred between processes. Although an intermediate code file could be produced to store compiled code, it would have to be placed in a temporary directory and deleted after the program is run, canceling out much of the benefit of using such a file. To keep things simple the interpreter simply traverses the syntax tree processing the code directly as it runs. This means that the language is strict rather than lazy, but this makes the internal garbage collection much simpler.

The external interface exposed by the library to the host language consists of two main functions, \texttt{Minimal\_function} and \texttt{Minimal\_callFunction}. The first of these two functions takes a chunk of source code, parses it and prepares it to be run. It returns a handle to the function which can be passed around as a value. The code does not need to contain definitions for every function as names are looked up
int fib_num = 5;
Minimal_function f = Minimal_function(
    "fib: int -> int;"
    "fib x = if x == 0 then 0"
    "      else if x == 1 then 1"
    "      else fib(x-1) + fib(x-2);"
);

Minimal_value r = Minimal_callFunction(f, Minimal_fromInt(fib_num));
int i = Minimal_toInt(r);

Figure 5.5: The Fibonacci sequence implemented in MINIMAL inside C

fib_num = 5;
f = Minimal.Function("""fib: int -> int;
    fib x = if x == 0 then 0
      else if x == 1 then 1
      else fib(x-1) + fib(x-2);
"""
);
i = f(fib_num);

Figure 5.6: The Fibonacci sequence implemented in MINIMAL inside Python

at runtime, as is the case with type checking. The second function is used to call
the function. It takes a set of parameters and returns the value returned from the
function. As well as being able to load code for a single function similar functions
exist, Minimal_loadCode, Minimal_loadCodeFromFile, which takes code for many
functions, either from a string or a file.

The code given in figure 5.5 loads the fib function, then retrieves a reference to
the function before calling it. C’s simplistic type system forces the code to be verbose
and does not allow the MINIMAL language to be cleanly integrated. A preprocessor
could be used which would take combined C/MINIMAL code and output C code.
This is only sugar which, while useful to the programmer, provides no additional
functionality and is therefore left as future work.

In a more flexible language, such as Python, the MINIMAL language can be
integrated much more closely with the host language as shown in figure 5.6.

5.4.4 Mobile Code

In order to transfer functions between processes the MINIMAL library provides a
function which retrieves a function’s code, plus the code of every function it calls.
This complete function tree is returned as a string containing XML, and contains
all the information necessary to load and run the code on a different machine. The
MINIMAL library also contains a function for loading this XML into an new ‘layer’.
These layers prevent functions retrieved as part of a closure from overwriting func-
tions already defined on the local machine with the same name. They can be thought
of as a module system but where the functions are assigned dynamically to modules. All functions retrieved from a LINDA tuplespace as part of a function tree are assigned to a new, unique layer. The value returned as part of the tuple is a reference to a function in this unique layer.

One obvious use for functions in a LINDA-like system is to allow a ‘master’ process to control a number of ‘slave’ processes by giving each of the slaves a function which they execute. By updating or changing the function the ‘master’ can control what the ‘slave’ process does. This results in the ‘slave’ running a fetch-execute cycle. When fetching a function from a tuplespace a new layer is created to store the function tree but the programmer is not given a reference to this layer. Instead the reference to the layer is placed inside the function to which the user gets a reference. In this use case the programmer will be repeatedly retrieving functions, creating a large number of layers to which they have no reference. Garbage collection of redundant layers is clearly essential. Each function is associated with each layer which it references. This is then used to perform reference counting-based garbage collection on the layers.

5.4.5 Isomorphism Wrappers

One of the reasons for using a single language was to overcome the difficulties that would occur in implementing conversion functions between different representations of functional values. The functions placed into a tuplespace by a process will be expecting a specific representation of a type, i.e. the representation used by the process that created the function. It is possible that the function will be retrieved by a process that uses isomorphic, but not identical, representations of the types — a function $f : a \to b$ may be retrieved by the type signature $f' : a' \to b'$. When the process which retrieved the function calls it, it will use a value of type $a'$, but the function is written to expect values of type $a$. There are three options for solving this. Firstly, the isomorphism algorithm could modify the function’s code to work with the representation of the type it has been matched against. Secondly, the algorithm could be included in the client programs as well as the kernel and used to convert values between representations as and when needed. Thirdly, the kernel could ‘wrap’ the function in another function which converts values between the different representations before passing the value to the main function, and then converting values back again. Modifying the function code would be a complicated process, especially as the algorithm would need to ensure that the semantics of the function were unchanged. The second option is easier, but the isomorphisms only need to be calculated once, when the function is matched therefore including the isomorphism algorithm in every client process is excessive. The final option requires the kernel to output the conversion functions in a format suitable for use as a Minimal function but as it must already be able to create functions which convert values within the kernel, it is not a big leap to create functions which can perform the conversion in a client process.
One issue is that the function may be directly, or indirectly, recursive. As the
code of the functions is not touched, it must be possible for the process which
retrieves it to call the function using the type representations it was matched against,
as well as being possible for the same function, or other functions, in the function tree
to call it using the original type representations. In order to allow this the conversion
code is placed in a function with a unique name, while the original function’s name
is unchanged. The new function is added to the original function’s closure, and is
set as the function which will be retrieved. When a process retrieves the function it
actually retrieves this new function, which when called alters the values passed as
parameters and passes them onto the original function.

Consider the function below, which takes the first element of a tuple and doubles
it.

\[
\begin{align*}
f &: (\text{int},) \rightarrow \text{int}; \\
f\ x &= x[0] \times 2;
\end{align*}
\]

If another function were to match that function using \(g :: (\text{int},()) \rightarrow
\text{int};\) the kernel would determine that the type specifications of the functions do
match, but that the parameters are not identical. In order to convert values passed
as parameters of the functions into the values of the type representations the function
is expecting, the kernel will create the following function:

\[
\begin{align*}
g &: (\text{int},()) \rightarrow \text{int}; \\
g\ x &= f((x[0],));
\end{align*}
\]

The function \(g\) simply creates an argument of the correct format, discarding
the unneeded second element of the tuple. When the process which retrieves the
function evaluates it, it calls \(g\) rather than \(f\) directly. Although this example does
not show it, similar processing occurs on the return value.

### 5.4.6 Matching By Value

Above it was described how functions must be wrapped, when matched against an
isomorphic, but not equivalent, type specification. In LINDA it is possible to match
against a value rather than just a type specification, and in this section what this
means for functions will be discussed.

Consider two functions which double a number.

\[
\begin{align*}
\text{double} &: \text{int} \rightarrow \text{int}; \\
\text{double}\ x &= x \times 2;
\end{align*}
\]

\[
\begin{align*}
\text{double2} &: \text{int} \rightarrow \text{int}; \\
\text{double2}\ x &= 2 \times x;
\end{align*}
\]

It is clear that these functions are equivalent, but they are not identical. If
the function \texttt{double} is matched against the type specification from \texttt{double2} then,
obviously, it will match. If the value of the function \texttt{double} is matched against the value of the function \texttt{double2} it is not clear what should happen.

The ideal solution is to determine the functional equivalence of the two functions, i.e. if they perform identical tasks then they match. Unfortunately, determining this equivalence to solving the halting problem, and therefore must be discarded. If the ideal solution cannot be used then not allowing matching by value for functions has to be considered. This would be an unpleasant irregularity for any implementation, as all values except a function value could be used in a \texttt{LINDA} template.

Functions are transferred by sending their abstract syntax trees between processes and the \texttt{LINDA} kernel. The obvious remaining solution is to test to see if the syntax trees are identical. Although this results in the two functions above not matching, it is better that they not match than having no functions at all match.

A common use case for matching by value is to remove a tuple from a tuplespace which has been read by other processes when matched by the type signature. A process \texttt{outs} a tuple, other processes read it, then the process performs an \texttt{in} using exactly the same tuple as the template. This ensures that the tuple that was \texttt{out}ed, and only that tuple is removed. In this case matching based on the syntax tree will succeed, and this use case is still possible.

5.5 Conclusion

In this chapter it has been shown how the fact that functions can be evaluated rather than being passive objects poses a challenge for using them in an open coordination system. It has been discussed why some of the principles behind the work proposed in this thesis have to be compromised to allow functions to be implemented. Chapter 7 shows several complete examples to further illustrate this point.

The various challenges faced by allowing any language, compiled to any machine code, to be transferred and run on a different machine have been set out. These challenges are very difficult to overcome therefore an alternative needed to be sought. A minimal, though Turing complete language, was introduced which would enable functions to be used in an implementation of an open coordination system.

The problem of matching function values was presented, along with a number of possible solutions. The best of these was to determine if the two functions are functionally identical, but this is equivalent to solving the halting problem and was therefore discarded. The possibility of not allowing matching by value for function was discussed, but the impact this would have on types which include functions was deemed too great. Instead a less ideal, but practical solution, of matching on identical parsed source code was chosen. This means that differences in the layout of the code and comments does not stop functions from being equivalent, but that differences in the parsed source code will.

The description of the \texttt{MINIMAL} language intentionally focused how the language can be used and its integration with other languages. Details of the semantics are largely unimportant for the purposes of this thesis but are included in appendix B.
for completeness. The type system of the language is the same as that described in the previous chapter and it is also discussed in the same appendix.
Chapter 6

Integrating With Linda

Previously it has been shown how a type specification language and a type isomorphism algorithm can match and convert between equivalent representations of types. In this chapter this will be applied to an open LINDA implementation. Initially the key features of an open implementation of LINDA will be described in order to ensure that this behaviour is maintained. Two possible methods for transferring type information from client processes to the LINDA kernel, and between kernels, will then be introduced and compared.

In many cases throughout this chapter the caching of type isomorphisms will be discussed. Just as important as caching the positive result of generating an isomorphism is the caching of a negative result, where the algorithm failed to generate one. In most cases with respect to caching of isomorphisms, the term ‘isomorphism’ is used to mean both an actual isomorphism and a value representing the failure to generate one.

6.1 Linda Requirements

An open LINDA system can be considered to have a number of important behavioral characteristics. These characteristics make LINDA the interesting and useful system that it is and should be maintained as much as possible by any extensions or alterations. One half of the characteristics are the appearance of the LINDA system from the perspective of a client process while the other half is the behaviour of the kernels themselves.

A client process interacts with a single LINDA kernel and knows nothing about any other kernels that may exist, or other processes that are part of the network. The interface that a client process uses is extremely simple and consists of a small number of functions for moving tuples into and out of tuplespaces, and creating new tuplespaces. Any changes to the system should, as far as possible, keep the interface simple. Previous chapters have described how to convert values from their in-memory representation to a format capable of being transferred over a network. Although this does add to the complexity, it is unavoidable as the type system is being made to support significantly more complicated types. When embedded into a
language that supports automatic type conversions, or uses dynamic typing, the use
of compound types in tuples can become transparent, apart from the serialisation
required, which was discussed in chapter 3.

The key requirement of a LINDA kernel is that it presents the interface expected
by the client processes, without any interaction from a user. The kernel must be
able to find tuples that are stored on any other kernel, and transfer tuples to other
kernels that request them. The 'open' feature of an open system is that processes
and kernel nodes can enter or leave the network at any time. A kernel must be
able to cope with this, and not lose information or to leak memory unrecoverably.
The work is presented as an extension to an existing LINDA kernel therefore any
additional features of the kernel, such as fault tolerance, should be maintained.

The primitive operations used by clients to communicate with the LINDA kernel
have a semantics which must be maintained. An \texttt{out(t)} operation has the simplest
semantics. It takes the given tuple and places it in a tuplespace. This action is
asynchronous, it does not matter whether, when the function returns, the tuple
has finally reached the tuplespace, only that it eventually will. More complicated
semantics for \texttt{out} have been defined, such as [122]. Given the semantics for \texttt{inp} and
\texttt{rdp} from [37] that are used in the implementation of this work the simple definition
of \texttt{out} is all that is required. The two input primitives, \texttt{in(t)} and \texttt{rd(t)}, are
both synchronous operations which take a template and return a matching tuple.
The \texttt{in(t)} operation is two operations on the server which outwardly appears to
be a single operation. Firstly a tuple matching the template is found, secondly
the matched tuple is removed from the tuplespace. If the template is successfully
matched then the tuple it matched against must be removed and returned in a
single operation. A non-destructive read does not require the tuple to be removed
and therefore consists of a single operation.

6.2 Approach

In a fully distributed implementation of LINDA, tuples for a single tuplespace can
be spread across a number of kernel nodes. In each of the partitions of a tuplespace
each tuple is stored, at least abstractly, as a sequence of value/type pairs. In the
case of an implementation that relies on the host language for type matching, the
type part of the pair is stored internally by the host language, as part of the value.
With more complex type information that is not specific to any one language, the
type information must be stored explicitly. Type information that is separate from
the built in language’s types will use additional memory, especially for the more
complex types. It is clear that memory can be saved by storing one copy of each
unique type and storing references to that rather than storing it multiple times.
Caching type information like this also leads naturally to caching the generated
type isomorphisms, which will be discussed in more detail later.

Systems that rely solely on atomic types send full type information with each
element of any tuple being transmitted. As the number of possible types is small,
the complete type information can be reduced to a single number which points to an element in the list of possible types. As the type information is small and fixed this is a natural solution to the problem of telling the kernel which types are used in each tuple. When sending more complex type information this method need not be changed, and the complete type information can still be sent with every element, in every tuple. An alternative is to send the complete type information only once and for the kernel to return a unique identifier that can be used in place of the larger type information when sending tuples. Each of the these methods will be discussed in turn below, and then additional changes to the Linda system, which are needed by both methods, will be described.

Both methods provide the same interface to the programmer, as the requesting of a unique type id from the kernel will be hidden from the programmer. As the two methods do not change the external view of the Linda system the criteria for deciding between the two strategies will be their runtime performance, impact on the scalability and fault tolerance of the system, and the ease with which they can be implemented. If one is significantly slower than the other then that will have a significant disadvantage. Any adverse impact on the scalability or fault tolerance of the implementation being extended will also count against the strategy.

6.3 Unregistered Type Information

If type information is transmitted as and when needed then the protocol of communicating between a client and the kernel, and between kernels is no more complicated than before complex type information was introduced. As described in chapter 3 type information is initially provided by the programmer as a string. This information is then parsed and used in constructing the isomorphism. If the complete type information is being sent with every value, and is sent as an unparsed string, then the server will need to parse it each time a value is sent. This will put an unnecessarily large strain on the server, therefore a partially parsed format is used. The client application parses the type specification provided by the programmer to confirm it is valid, the syntax tree is then serialised when transmitted to the server. In the implementation provided with this thesis the serialised format is XML based, but any format could be used. The schema for the XML format is given below.

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://code.google.com/p/pylinda"
    xmlns="http://code.google.com/p/pylinda"
    elementFormDefault="qualified">

    <xs:element name="typeobj">
        <xs:complexType>
            <xs:choice>
```

129
<xs:element name="nil" type="nil"/>
<xs:element name="id" type="id"/>
<xs:element name="type_function" type="type_function"/>
<xs:element name="product_type" type="product_type"/>
<xs:element name="sum_type" type="sum_type"/>
<xs:element name="ptr" type="ptr"/>
</xs:choice>
<xs:attribute name="name" type="xs:string" use="required"/>
<xs:attribute name="typeid" type="xs:string"/>
</xs:complexType>
</xs:element>

<xs:element name="nil"/>

<xs:element name="id">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="typeid" type="xs:string"/>
  </xs:complexType>
</xs:element>

<xs:element name="type_function">
  <xs:complexType>
    <xs:sequence>
      <xs:choice maxOccurs="2" minOccurs="2">
        <xs:element name="nil" type="nil"/>
        <xs:element name="id" type="id"/>
        <xs:element name="type_function" type="type_function"/>
        <xs:element name="product_type" type="product_type"/>
        <xs:element name="sum_type" type="sum_type"/>
        <xs:element name="ptr" type="ptr"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="product_type">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="nil" type="nil"/>
        <xs:element name="id" type="id"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
There are several different paths that type information could take when tuples reach the kernel. The simplest is that the stored tuples continue to contain pairs of the complete type information and the value. In this case type information is compared only when type matching is explicitly required. This has the advantage that no work is performed before it is required and it will be the simplest to implement as the only significant change between this and a traditional LINDA implementation is in the matching algorithm.

It is obvious that types are likely to be shared by more than one value. Rather than store the type information with each value the duplicate type specifications can be coalesced into a cache, either per-tuplespace partition or as a single cache for each kernel. A cache of type information is a mapping \( type \ id \rightarrow type \ object \). If
this caching is done per tuplespace some, or all, of the possible type isomorphisms can be pre-calculated. As the per-tuplespace cache would contain all of the types that are currently in use by values inside the tuplespace then only tuples containing only those types could possibly be successfully matched. If a template containing a type not in the cache is matched then it can be discarded before a single tuple is examined. If, rather than have a cache for each tuplespace partition, there is a single cache per kernel then memory usage will be further reduced as only one copy of each type specification will ever be stored in each kernel.

The only disadvantage for a per-tuplespace cache compared to a global one is that types used in multiple tuplespace partitions will be stored more than once. This will clearly impact on memory usage as the additional memory required for each tuplespace will increase for each additional tuplespace that is used, regardless of whether any new types are used. To achieve the best of both worlds a global cache can be used which maps the type specification to a unique identifier. This identifier is then used in a type cache for each tuplespace partition.

Transferring types between client processes and a kernel is half of the problem. In a multi-kernel system, kernels will need to transfer tuples between each other. In an unregistered type system complete type information is sent with every tuple that is communicated between kernels, just as it is between clients and kernels. The fact that a kernel may cache type information is largely irrelevant as kernels must take information out of the cache when a tuple is transferred. This keeps the number of messages to a minimum, but increases the relative size of each one.

The final stage of a tuple’s journey is when it is returned by a kernel to a client. In this case no type information needs to be transmitted as the client already knows the types of the values contained in the tuple. Whether a client is communicating with the kernel using a single thread, or many, is not important as each connection is blocked, awaiting at most a single tuple. In the case of the predicated primitives the kernel will return a special value if a deadlock was detected. This tells the client to unblock itself, and not to wait for the tuple it was expecting to receive.

In order to achieve the best performance a LINDA kernel should process a tuple as little as possible. Tuples are the pieces of data that are used the most in a LINDA system and anything that has to touch each tuple will have to do so many thousands, or even millions of times. Tuple scanning [44] is probably the worst thing that could be done to a tuple as it involves processing every item of data in every tuple, a task that will have to be repeated so often that even the quickest process will have an effect on the performance of the system. Caching the type information will require tuples to be scanned, as garbage collection of the information and pre-calculation of isomorphisms (see section 6.6) relies on accurate counts of the number of data items using each type being kept. In order to reduce the cost of this scanning it should be combined with other scanning which is required.

Consider two processes running on two separate machines, with a LINDA kernel running on each. Process A outs a tuple containing a string and a pair of 32 bit
integers into the uts. The library being used by the process will construct a message containing the string and the string type, the value of the pair of integers and the type object for the pair of integers. When received and decoded the kernel compares the received string type against all the types currently stored in the type cache\(^1\). If it exists the same type id would be used, but as the kernel has no other types stored it is given the id 11. The mapping \(11 \rightarrow \text{string type}\) is added to the type cache and the type object is replaced with the type id in the first element of the tuple. The process is repeated for the second element of the tuple, with the type for the pair of integers being given the type id 12. The tuple is added to the uts tuplespace.

Process B constructs a tuple containing a string type and a type representing a pair of integers. The process makes an in request to its local kernel. The kernel compares the string type to its type cache and discovers that such a type does not exist. The type is added and given the type id 21. The kernel then attempts to match the type to other types in the uts using isomorphisms. As the kernel is empty this fails. The kernel then forwards the entire in request to the other kernel. The other kernel compares the string type to the types currently stored and finds it is identical to type 11, then it compares the pair of integers type and finds it is identical to type 12. An isomorphic match with the typed tuple \((11, 12)\) is then sought, and tuple outed by Process A is found. It is removed from the tuplespace and the type information is stripped. It is then sent back to the requesting kernel, which passes the message on to the requesting process. The type information is added back to the tuple (based on the types in the request) and the tuple is returned to the calling code.

6.4 Registered Type Information

In the previous section the traditional method for transferring type information with tuples was extended to use complex type information. Rather than transfer the complete information, which will be repeated often, wasting valuable network bandwidth, the client process can ‘register’ a type with the kernel. The kernel stores the type specification and gives the client a unique identifier which can be used to identify that type in all future communications between the kernel and the client. This has several advantages, not least that the size of the unique identifier is likely to be much less than the size of the complete type information. A 32-bit integer would be sufficient to provide a unique identifier for all types registered with a kernel as it is highly unlikely that more than four billion distinct types would be registered with a kernel at any one time. Only a type specification of less than four characters would be smaller than this identifier, and when this saving is multiplied by the number of elements in a tuple, and the number of tuples then the saving becomes very significant. The registration of types also allows the kernel to easily pre-calculate type isomorphisms and dynamically alter the structure that tuples are

\(^1\)This case does not involve isomorphisms, rather it is looking for identical types
stored in based on the type information.

As was discussed in a previous chapter a client process must create a ‘type object’, either explicitly or implicitly from the programmer’s perspective. When the type object is created a message can be sent to the kernel containing the type specification. The kernel compares the specification to others already stored in the registry. If it matches then the identifier already assigned is returned\(^2\). If the type specification is new then the kernel calculates a unique identifier and stores this specification in a registry with the id. Each kernel must be capable of generating globally unique identifiers that can be returned to a connected process when they register a type. A similar mechanism must already be in place for creating tuplespaces as internally each tuplespace will be given a name, even if this name is not visible to client processes. Adapting this to provide unique names for types is a trivial task.

Scalability is an important factor for any LINDA implementation. In order to help create a scalable system type information should only be stored where it is necessary. The kernel to which the client is directly connected must hold a copy of all the type specifications registered by the client as it is the client’s point of contact with the rest of the LINDA network. Tuples placed into the local kernel’s partition of a tuplespace may end up being transferred to other kernels in the system. The other kernels must therefore have a method of using types registered by a client process connected to a different kernel. This would require the type specification to be transferred as well. When a client registers a type with its local kernel this registration could be broadcast to every other kernel in the system. This would ensure that every kernel knows about every type in use within the system and therefore when values are moved between kernels it could be assumed that the types already exist on the destination kernel.

Transmitting all type information to every server in a LINDA network is not scalable. Even if the types are coalesced such that all identical types across all kernels are given the same unique identifier, every time a process registers a new type it will still result in a message being sent to every kernel in the network. Rather than register types across multiple kernels pro-actively the registration could be deferred until the moment when it is required that the type signature be known on another kernel. When types are registered they are given globally unique identifiers by the kernel where the type was initially registered, and these identifiers are used when transferring the values in tuples to other kernels. If a value is transferred to a kernel that does not have an entry for a particular unique identifier in its registry then it must find the appropriate specification by requesting it from another kernel. If, along with the tuple or template, the identity of the sending kernel is transferred then the receiving kernel will know at least one kernel who has access to the specification. The kernel must store the tuple in a ‘holding area’ while it requests the appropriate type specification from the sending kernel. Once that kernel has responded the tuple

\(^2\)This is not strictly necessary as the isomorphism algorithm would declare the type specifications as equivalent anyway. It is better to avoid using the isomorphism algorithm and instead perform a cheap one time equality check on the type registry.
can be released from the ‘holding area’ into the appropriate tuplespace partition. If
the tuple is placed into the tuplespace immediately then a different request to the
kernel could result in the types in the tuple being matched against, before the type
signatures have become available. It is also possible that the sending kernel could
crash before the receiving kernel has time to request the type’s signature. As type
signatures are not being pro-actively transferred between kernels the crashed kernel
may have been the only kernel which knew the type specification. If the system
is fault tolerant then it must be resistant to failures such as this. By storing the
tuple in a ‘holding area’, if a crash occurs before the operation is completed then the
kernel can act as if the tuple never arrived in the first place and the fault tolerance
mechanisms invoked. If a user cleanly shuts down a kernel then it must wait until
each tuple transfer has been completed, including any transfers of type specification
as a clean shutdown should not invoke any fault tolerance mechanisms.

The algorithm followed by a kernel upon receipt of message from another kernel
is as follows.

1. Does the message contain a tuple? If yes go to 2, if no go to 3.

2. For each element in the tuple:
   (a) Get type id for element.
   (b) Is the type represented by the type id in our type cache? If yes move to
       the next element of the tuple, if no go to 2 c).
   (c) Request the type object for the type id from the kernel making the request
       and add it to the type cache.

3. Process the message as normal.

In the case of the two bulk tuple operations, collect and copy_collect, there
are two implementation options for a fully distributed system. Either the tuples
can be moved between partitions on the kernel where they were previously found
or they can be moved to the kernel which initiated the bulk tuple operation. In
both cases the movement between tuplespaces could involve converting between
different type representations, or the values could remain untouched. A common
use of collect and copy_collect is to move a group of tuples into a tuplespace,
then to retrieve them. In this case converting the values when performing the first
move will mean that when performing the retrieval the isomorphism calculation has
already been done speeding up that operation. It is dubious whether this would
give any increase in overall performance. The first of these two operations is the
most efficient as it creates no network traffic beyond the initial request while the
other requires many tuples to be moved and may cause type specifications to be
transferred between kernels. As it is likely that following a bulk tuple operation a
process will subsequently make requests for the tuples it has just moved, moving
all tuples to the process’ local kernel may be advantageous. The choice of which

135
method to choose is left to the implementer and both are compatible with complex data types. The implementation described in this thesis chose not to move tuples.

If a kernel node is gracefully leaving the system while others are still connected, and its tuplespace partitions are not empty, then it must transfer all those tuples to another node before exiting. As a tuple can be retrieved by any process, connected to any node, all of the tuples must remain in the system after the kernel has left. This bulk move of tuples must take place in a similar fashion to that described above for the bulk tuple operations which move tuples between kernels. The only difference is that it is the kernel that is sending the tuples which initiates the move, not the receiving kernel. The kernel which is terminating must not leave the network until it has had confirmation from all kernels to which it is transferring tuples that they have copies of all the required type specifications. It is possible that no process connected to the system has registered a type that is isomorphic to one used by a value stored on the leaving kernel. If this is the case it is tempting to discard this value as no current process could retrieve it. In an open system a process could connect at any time, or a currently connected process could register a new type that is isomorphic. Due to the open nature of the system it is therefore not possible to discard values based on their types.

The success of an open system also depends on the ability for it to run indefinitely with processes and nodes joining and leaving throughout its life. If a new type is registered with a kernel by a process then this will take up some memory in the kernel process, this is unavoidable. When a process disconnects any type specifications which are not currently in use by any values stored in that kernel, and have not been registered by any other currently connected processes, must be deleted in order to reclaim memory. If a value exists in a tuplespace partition on the kernel node then the type specification has to be retained. Should that value be removed in the future from the partition, and no other values of that type are left then the type specification must be deleted. In order to achieve this the type registry inside the kernel must store the process ids for processes that have registered each type, as well as a counter for the number of values which use the type. In order to keep the count accurate every tuple must be scanned on entry and exit to and from the kernel, and the count for type specifications updated.

It was described above how a kernel must ensure it has the type specification for each type used by a value that it stores. This was because for anything to happen to a tuple it must first be matched, a process that requires the type specification. A side effect of this is that kernels do not rely on any other kernel for type information. This means that ‘reference counting’ garbage collection of type specifications can be done entirely locally.

Consider again two processes running on two separate machines, with a Linda kernel running on each. Process A registers a string type with its local kernel by constructing a message containing the type object and requesting a type id. The kernel looks for identical types in its type cache, but as it is empty, finds nothing.
It gives the string type the globally unique identifier 11 and returns that to Process A, which associates the id with its copy of the type. The process is repeated for the type representing a pair of integers, which is given the id 12. Process A then `outs` a tuple containing a string and a pair of 32 bit integers into the `uts`. The library being used by the process will construct a message containing the string and the string type’s id, the value of the pair of integers and the id for the pair of integers. When received and decoded by the kernel the tuple is added to the `uts` tuplespace. No further processing is required as the types have already been replaced with unique identifiers.

Process B goes through the same type registering process as Process A, getting the type id 21 for the string type and 22 for the pair of integers in return. It then constructs a tuple containing a string type id and the pair of integer’s id. The process makes an `in` request to its local kernel. The kernel looks up the type objects based on the ids supplied in the request and attempts to match the type to other types in the `uts` using isomorphisms. As the kernel is empty this fails. The kernel then forwards the entire `in` request to the other kernel. The other kernel attempts to look up the type objects based on the ids in the request. This fails as the types are not in the kernel’s type cache. It requests the type object from the kernel which forwarded the request on, and adds it into its type cache. An isomorphic match with the typed tuple (21, 22) is then sought, and the tuple `outed` by Process A is found. It is removed from the tuplespace and converted from type (11, 12) to type (21, 22) which in this case is the identity conversion. It is then sent back to the requesting kernel, which passes the message on to the requesting process. The complete type information is added back to the tuple (based on the types in the request) and the tuple is returned to the calling code.

6.5 Comparison

The two methods described above present an identical interface of the system to the programmer. The differences between the two methods are hidden in the library used to communicate with the kernel, and the kernel itself. In the comparison of them the memory usage, network usage and overall performance will be considered.

An unregistered system with no type caching will use the most memory of all the options. Storing the complete type information for every value will quickly increase the memory used as the number of tuples stored increases. A tuplespace specific cache will decrease the amount used, but a global type cache will use the least memory. A registered system has the problem that all types which may be used, whether they are actually used or not, must be registered. These used types will take up some memory in the kernel, but it is unlikely that many types will be registered and not used so the amount of wasted memory is likely to be minimal.

With respect to the network usage the registered system will consume far less bandwidth. A typical globally unique identifier is a mere 16 bytes in length, and could be as little as 4 bytes long. A 16 character string will not allow you to specify
a very complicated type, especially as a minimum of 3 characters are used to specify the name of the type and delineate the specification. In an unregistered system the type identifier need only be unique across a single kernel so a four byte integer would suffice. It is hard to imagine a case, other than deliberate stress test, where a single kernel node would encounter more than 4 billion distinct types at any one point in time. Once all values and processes that reference a type specification have left the system the type can be unregistered and the identification value used again. As type names are used purely in the context of a single type specification the number of distinct types could be reduced by treating types that are structurally identical as the same type in the cache. Figure 6.1 shows how two types, which share an identical structure could be given the same identifier. This has the disadvantage of requiring a more complicated method for assigning values to and retrieving values from the registry, but it provides another method which could reduce the memory usage.

```
Type Registry = {}
Process 1 -> Register Type 'a: Nil + (int * a);' -> 1
Type Registry = {1: 'a: Nil + (int * a);'}
Process 2 -> Register Type 'b: Nil + (int * b);' -> 1
Type Registry = {1: 'a: Nil + (int * a);'}
```

Figure 6.1: Using A Nameless Type Registry

It was described earlier how a partially-parsed format could be used to transfer type specifications between the client and the kernel. The names used in the specification can be replaced with generic identifiers in this format removing the names altogether. To decide if two structures are identical a simple equivalence test can be used.

Network bandwidth usage is especially important between kernel nodes as they are inter-node connections whereas a client-kernel connection is intra-node. Inter-node connections are likely have a much higher latency and much lower bandwidth than connections between processes on the same machine. An unregistered implementation will use considerably more bandwidth as the full type information, consisting of many bytes, will be transmitted for each and every element of every tuple.

The implementation of an unregistered version is simpler than a registered version due to the smaller number of changes from a traditional implementation. If no type caching is implemented then only the protocol for communicating between clients and kernels, and between kernels, needs to be extended to cope with the extended type information. The type matching algorithm that is the core of a tuplespace also needs to be updated, but those are the only two changes to the functionality that are required. If type caching is implemented then more changes are required. However, they can also be restricted to a few well defined locations. If the type cache is for all tuplespace partitions in a kernel then at the point where tuples
enter, or leave, the kernel they must be scanned and the type information is either entered into, or retrieved from, the cache. A tuplespace specific type cache will need to perform the tuple scanning at the interface between the kernel and tuplespace partition.

A type cache for an unregistered system not only reduces the memory cost of storing tuples, but it also opens the door to caching type isomorphisms and even pre-calculating matches while the server is idle. Any of these extensions would increase the complexity of the changes, and are not specific to an unregistered system but could also be applied to a registered system, and hence will be discussed below.

A registered version does not need to perform as much type scanning as an unregistered system as the caching and coalescing of identical types is performed explicitly through the registration of specifications before they are used. When a type object is created the library used to communicate between the client and kernel needs to request a type identifier. This is a one off operation for each type registered by a program and therefore adds little overhead. A typical program will only use a small set of types that are registered before any real work is done. A pathological test case might continuously register new types, and use them once before discarding them but it is hard to imagine why a real program would operate in this fashion.

Results for the experimental evaluation of both implementations, which confirm these predictions, are given in chapter 8.

6.6 Optimisations

The biggest cost, by some distance, in the use of type isomorphisms is the generation of the isomorphisms themselves. The experimental results in chapter 8 will show that this cost has by far the biggest effect on the system’s performance of any change proposed in this thesis. The cost of generating an isomorphism is very high, especially when compared to the cost of performing a direct equality test on simple atomic types. Comparisons between types are highly likely to be repeated a large number of times. In order to reduce the cost of the test as much as possible, the generation of each isomorphism must only be performed once. In order to achieve this a kernel-wide cache of isomorphisms is used. This cache contains entries for pairs of type representations and each pair is mapped to either the isomorphism or a value indicating that they do not match. The cache is faster than generating the isomorphism because it uses equality testing on the types, or hashing, rather than the more expensive isomorphism.

Caching type isomorphisms (or the failure to generate one) is not a straightforward matter as there is a danger that it could impact on the scalability of the system. A naive implementation of a caching algorithm would permanently store the isomorphism for every comparison it has ever performed. If processes disconnect from the kernel node then type isomorphisms may still be cached when no process uses either of the two types. This leads to a growth in memory usage that does not decrease as processes leave the network. Two methods can be used to reduce the
impact that an isomorphism cache has on the memory usage of the system over time. Firstly the cache can be based on the actual data stored in the tuplespace partitions on the kernel, or secondly the cache can be limited to a set size, and isomorphisms discarded to prevent the cache from growing too large.

The first of these two caching methods has the distinct advantage of caching as much as possible, but not caching for longer than necessary. There must be a single type isomorphism cache for each kernel, otherwise the same comparison may be performed more than once in two different tuplespace partitions. The cache consists of two elements, the cache and a type counter for each tuplespace partition. The counter must be updated as values pass into and out of the tuplespaces. When a tuple enters, the counters for each of the types, of each of the tuple’s values, are incremented. When a tuple leaves, each of the the associated counters is decremented.

When the counters for a particular type reach zero across all tuplespaces then there are no values of that type left in the kernel. In an unregistered system the only options are to either discard any isomorphisms involving this type immediately, or to discard them after some timeout. It is possible that a value using a type could be added after all other values of the same type have been removed. If the isomorphisms are immediately discarded when all the values are removed, then the addition of a new value of an identical type will cause the previously cached isomorphisms to be recalculated. In an attempt to avoid this the isomorphisms could remain in the system for a period of time after the last value is removed. If no value of that type representation is placed into a tuplespace during this period then the isomorphisms are discarded. It is impossible to answer the question of how long the timeout should be as an unregistered system has no knowledge of which processes are using which types and thus the timeout must either be hard-coded or derived using some heuristics based on the system’s previous performance.

If the counters reach zero for a type in a registered system it is known which processes use which types. Even if all the values using a given type have been removed then the cached isomorphisms can be kept as processes still exist which could add them to the system again. Of course in an open system a new process could join at any moment and register a type that was discarded, but this is no different from an unregistered system and at least in this case the system knows whether any currently connected processes can use a given type. In order to support this the cache must be integrated into the type registry such that the cache can tell how many processes are still registered to use a given type. The type registry must also notify the cache when the last process which had registered a particular type leaves the system, soo that if no values of that type remain in the system then type isomorphisms can be removed from the cache.

Many programmers will not use a single, large type specification entirely built up from atomic types. Instead they will use a number of types that reference each other (examples of this were given in chapter 4). This type hierarchy must be taken into account when updating the counters in the registry (or equivalent type
The fact that sum types are supported means that some types may be referenced by other types, but are not necessarily included as part of a value. Recursive types also pose a problem if the count is to be accurate as it is the value that will have to be scanned to update the counters, not the type. As the value will change each time and yet the type remains constant this will be difficult to optimise.

The Linda kernel will never split a value into pieces and although a value may be converted between isomorphic types the value itself remains whole. As a result of this the additional values in recursive types, which are referenced by the top level value, can be ignored, and only the first value of any type counted. Scanning a tuple or template is a relatively expensive process, especially as it has to be done for each and every tuple that enters or leaves the kernel. Rather than adding more to this already expensive process the complete set of types referenced by a specification, including those referenced indirectly, can be calculated when the type is registered. This list of types is stored in the registry and then whenever a value of a type is added or removed this list is used to adjust the referenced types’ counters as well.

This has the disadvantage of treating all elements of a sum type as existing in every value, which can lead to types being stored for longer than absolutely necessary. The expense of calculating these counts exactly (by scanning each and every value to determine which element of a sum type is used) is prohibitive, and the types will eventually be deleted when the values of the parent types are removed.

### 6.6.1 Preemptive Isomorphism Calculation

If a request is made to match a given template against the tuples currently stored in a tuplespace partition and one or more of the types in the template have not been matched before then there will be a high initial cost for processing this request. Subsequent uses of the same or similar templates will have a much lower cost due to the caching of the generated isomorphism. A likely structure for a process connecting to a Linda network is that it will connect, register the types it will use and then begin to perform its work. In the gap between the registration of the types and the beginning of its work the kernel could calculate the type isomorphisms involving the types registered by the process. If the kernel is idle then using this time to calculate isomorphisms that will be needed in the future will reduce the cost when the types are first used.

A danger of preemptively calculating isomorphisms is that too many will be calculated. The time in which calculation of isomorphisms should be done is limited as it must only be done during idle time for the kernel, otherwise it will interfere with the handling of requests from client processes or other kernels. In order to ensure that the most benefit is achieved during the idle time the isomorphisms need to be calculated in an order from the most likely to be used to the least likely. Calculating which types are most likely to be used is predicting the future usage...
patterns of a process, when in an open system the kernel has no knowledge of a
process before it connects. The kernel keeps track of which processes have access to
which tuplespaces, and types that belong to processes which do not have access to
the same tuplespaces cannot be compared. Pre-calculating isomorphisms for these
types would be a waste of time. Initially every process has access to the \textit{ut}s,
but by retrieving values containing references to other tuplespaces, or creating the
tuplespaces directly, the process can gain access to other tuplespaces.

As every process has access to \textit{ut}s, performing the pre-calculation of isomor-
phisms based on processes sharing access to tuplespace would reduce to the general
case of inferring them all. If \textit{ut}s is ignored and isomorphisms are only calculated
between types of two or more processes that share access to the same tuplespace(s)
then the set of isomorphisms that need to be pre-calculated can be reduced, but will
still cover all possible cases that can currently match. If we assume (temporarily)
that all client processes will register all the types that they will use before they start
to transfer values, then no pre-calculation can take place as all processes will only
have access to \textit{ut}s and calculating based on this tuplespace has already been dis-
counted. A common use case for a LINDA program is to construct a new tuplespace
and perform all interaction with that and just use \textit{ut}s to communicate a reference
to the new tuplespace. Each time a process gains access to a new tuplespace its
types can be considered for pre-calculation with the types of other processes which
already have access. Removing the assumption that a process registers all types
before transferring values we can see that each time a type is registered a new set of
isomorphisms will need to be considered for pre-calculation based on the tuplespaces
that the process currently has access to.

Taking all that has been discussed above the following algorithms implement an
effective form of caching and pre-generation of isomorphisms.

\textbf{Process Registers A New Type}

1. Check type registry to see if an identical type exists, if YES then return that
type's identifier and skip to step 3, if NO then continue

2. Create entry in type registry and return new identifier

3. Get list of tuplespaces accessible to process

4. For each tuplespace add a sequence of pairs containing this type and each type
   already linked to the tuplespace to the pre-calculation queue.

\textbf{Kernel Becomes Idle}

1. Order pre-calculation queue by the sum of count of values for both types in
   the pairs

2. Take the pair with the highest count and calculate isomorphism

3. Enter isomorphism into the cache

142
4. If kernel is still idle continue from 2.

Much of the success of the optimisations depends on idle time being used effectively by calculating isomorphisms that will be required in the future. If the pre-calculation is done while the computer that is running the kernel is not idle then the overall effect could be to slow the system down. The operating system will need to divide the processor time between the client which is doing useful work, and the kernel which is calculating an isomorphism that it hopes will be used. It is reasonable to assume that the operating system will schedule the Linda kernel and any other processes fairly, and so the kernel need only balance the ‘eager’ generation of isomorphisms with its normal request processing.

### 6.6.2 Garbage Collection

When multiple tuplespaces were added to Linda, garbage collection needed to be introduced to ensure that tuplespaces which no active process, or any future process, could access would be removed from the system. Garbage collection is performed using reference counting based on who ‘owns’ tuples containing a reference to the tuplespace or which processes have a reference to it. The owner of a tuple can be a tuplespace or, if the tuple is currently being processed, a kernel. In order to ensure that the reference counts are updated correctly tuples must be scanned for tuplespace references before they are moved between tuplespaces, kernels or processes. In a traditional system tuplespace references could only be found if they were items in a tuple. If the programmer had used their own encoding scheme to fit complex types into an atomic type-based system then garbage collection would break.

A system which allows compound data types to be used will allow tuplespace references to be included as part of a much larger value. The kernel, guided by the type specification, will know how to decompose the large value and therefore can track tuplespace references correctly. As well as examining every tuple a naive implementation would have to completely traverse each value. This would have a serious impact on the speed of the system therefore the frequency that the kernel is required to completely traverse a value needs to be reduced.

Earlier, garbage collection was discussed for the type specifications and it was decided that a list of referenced types would be stored with each type specifications. In addition to this list a boolean value is stored which indicates whether the type, or any of the types it references can store a tuplespace reference. By looking at this boolean for the type of each value in a tuple the kernel can quickly and easily see which values need to be traversed, if any. This value contains information about whether any of the types may contain a tuplespace because if this value is false then the kernel is guaranteed that the value does not contain a tuplespace reference. If the value is true then the kernel must traverse that part of the value that makes up this type, but it can check the tuplespace indicator for each of the types it references and therefore ignore portions of the value.
The information stored with each type specification tells the kernel only that a
value of that type may contain a reference, not that it actually does. A tuplespace
reference type may be included as part of a sum type and the actual value may
contain a different element of it. Rather than force the client to keep track of what
values are combined to make a complete value, a complicated process, and then
send this information with the completed value, the kernel will traverse the value
and may not find a reference.

In an unregistered implementation the kernel itself will parse the type specifi-
cations and then it must analyse the specification for the presence of a tuplespace
reference. A registered implementation on the other hand uses the client to parse
the specification and sends the parsed information to the kernel. The client will send
the kernel information regarding the tuplespace references when the specification is
sent. This fits with the two different strategies, more processing on the kernel with
larger and less numerous messages in the unregistered version but less processing on
the kernel with smaller and more numerous messages for the registered.

Garbage collection is vital to persistent LINDA systems therefore the scanning of
tuples for references to tuplespaces must be performed. The method proposed which
relies on precalculated dependency trees for types (rather than examining each value
individually) is a compromise between extra processing by the client and processing
by the kernel. By cutting down the traversal of values to only those which may
contain a tuplespace reference the cost of scanning tuples is reduced to a minimum.

6.7 Implementation

In this section the two methods for altering a standard LINDA system, given above,
will be implemented. Since the two versions use different strategies they will be
implemented separately.

This section only considers the implementation of the changes to the LINDA
client library and kernel program so they use type isomorphisms. Work previously
introduced involving the transmission of typed values and the generation of type
isomorphisms will be used extensively, but not described here. For more details on
these parts refer back to earlier chapters.

6.7.1 Unregistered

As the unregistered implementation is simplest it will be considered first. Initially
the simplest method of implementing it, as was introduced above, will be discussed
but in order to achieve a reasonable operating speed caching will also be implemented
and evaluated.

Client Side

The biggest change to the client side is the introduction of a type object, as was
discussed in detail in chapter 3. When created, the type object does not contact the
Kernel, instead it stores the type specification which is then included with every tuple transmitted to the kernel, which uses that type. Beyond that there is no change to the client process, the only other changes take place in the kernel.

**Kernel Changes**

Once a tuple has been transmitted from a client process to the kernel it consists of a sequence of type specifications and values. The simplest possible implementation does not touch the tuple and instead stores it intact in the tuplespace. The tuple matching algorithm is the same as before but rather than using an equality test it calls the isomorphism algorithm. These simple changes to the client library and kernel allow the system to perform everything that was set out in chapter 1. Compound data-types can be easily and simply used by programmers with none of the LINDA principles broken. The only requirement that has not been demonstrated is that the speed remains practical. This will be evaluated in chapter 8 using a series of ‘real world’ tests.

The first of the optimisations to be implemented is the coalescing of type specifications to reduce the excess memory usage through storage of redundant copies of identical specifications. The intuition for this enhancement is that the kernel can automatically register type information, rather than the client process doing it explicitly. The registration of types is internal to each kernel\(^4\), so no attempt needs to be made to make the identifier assigned to each type globally unique, it need only be unique across a single kernel. In this case an integer counter will suffice. Tuples and templates enter a kernel through a limited number of places. Most messages sent to and from a client will contain types, and a few inter-kernel messages will as well. For each of these locations in the code the tuples and templates must be scanned and each type specification either replaced with an id, or the id replaced with the specification depending on which direction the types are heading.

Type isomorphism caching needs to be implemented as the generation of the isomorphisms is a key factor in reducing the performance of the system. Rather than implement a complete cache, which would cache all isomorphisms at the expense of additional processing to ensure the isomorphisms are correctly discarded, a least recently used cache, will be used. The cache itself is extremely simple. A dictionary containing pairs of type specification identifiers mapped to one half of the isomorphism, or a value indicating no isomorphism was found. For each complete isomorphism there are two entries in the cache, one for each direction. In addition to the cache a list is kept of the uses of the isomorphisms to enable the least recently used isomorphism to be discarded.

\(^{4}\)In an unregistered system all communications between clients and kernels and between kernels must contain the full type specification.
Data Structures

In a kernel featuring auto-registration of types it must keep track of all the type specifications it has seen, and link them to its internal identifier. A hash on the textual representation of the type can be used as the key to store the specification in a dictionary. The usual caveats on hashing apply here as it is not guaranteed that a hash will be unique to a specification, although it is overwhelmingly likely.

On receiving a tuple the type specification for each element is hashed and compared with the mapping. If it exists then the same type identifier is given to the element in the tuple, if not a new identifier is created.

A mapping of type identifiers to specifications is also maintained while updating the mapping described above. This mapping is used before calling the type isomorphism algorithm to look up the type specification based on the unique id.

The type isomorphism cache is a dictionary that maps a pair of string identifiers of types to one of the isomorphisms. Python’s relaxed type system means that this sort of mapping is easy and natural to implement. All types inside the kernel have been assigned an identifier upon their entry into the kernel, so looking up an isomorphism is simple and quick. If the isomorphism does not exist in the kernel then it is generated and placed into the mapping.

To facilitate garbage collection of the cached type isomorphisms and specifications an extra data structure is maintained to make the garbage collection as quick as possible. This structure takes the form of a mapping from type identifiers to a structure containing two lists and an integer. One of the lists is used to store a list of all the processes that are directly connected to this kernel which have used this type. The first time a process uses a particular type it is added to the list, and when the process detaches from the kernel it is removed. While there is at least one process registered who has used a type that type is not garbage collected. The other list contains type identifiers which refer to this type. A type may not be directly referenced by a value, instead it may be included as part of another type. This list ensures that a type is only garbage collected when no other types refer to it. The integer is a count of the number of values that reference the type. Each time a value enters or leaves the kernel this value is updated. When the count reaches zero, and all other lists are empty then the type can be garbage collected.

Messages

The key advantage of an unregistered system over a registered one is that no new messages are introduced into the system, existing messages are simply augmented with expanded type information.

The XML schema for the type information was given in 6.3 and will not be repeated here. Values conforming to the schema are inserted into XML documents like the one given below.
The same format is used for both rd, out, collect and copy_collect messages. Other messages used between the client and kernel are purely for internal housekeeping and do not transfer typed values so will not be considered here. The tuplespace identifier used in the message is the internal identifier used by the system, and is not user visible. The id attribute to the value element is the ‘address’ of the value which can be pointed to by a pointer value in the value.

The example message shows a client performing an in operation on the tuplespace xyz. It is retrieving a two element tuple by providing a template with the first element as an actual value and the second as a type. The second element’s type is given by the first of the two typeobj elements. The second is a type that is referenced by type B.

The return message from the kernel consists of an XML document containing only a tuple element, or if the operation was inp or rdp a nomatch element.

The only inter-kernel message in the implementation that transmits typed data is the multin operation. This message is sent from one kernel to another when tuples are being moved. The message takes the form given in figure 6.2. In this message two tuples are being added to the xyz tuplespace.

6.7.2 Registered

The registered implementation requires more complex changes to the way the system operates, but greatly improves the efficiency of the system and results in faster communication between kernels due to smaller size packets (see chapter 8).

Client Side

When type objects are created they must contact the kernel to receive a type identifier for their type specification. It is possible that types will be created by the programmer before a connection to the kernel has been established. If this is the
<message>
  <action>multiin</action>
  <tuplespace>xyz</tuplespace>
  <tuple>
    <element>
      <typeobj name="A">...</typeobj>
      <value id="1234" type="A">...</value>
    </element>
    <element>
      <typeobj name="B">...</typeobj>
      <typeobj name="C">...</typeobj>
      <value id="5678" type="B">...</value>
    </element>
  </tuple>
  <tuple>
    <element>
      <typeobj name="D">...</typeobj>
      <value id="9012" type="D">...</value>
    </element>
  </tuple>
</message>

Figure 6.2: The XML message used to place two tuples into a tuplespace.

case then the request for the identifier is deferred until a connection has been established. When transmitting tuples and templates the type identifiers associated with each type specification are used in place of the type specification itself.

Kernel Changes

The type specification registry, a fundamental part of the registered implementation, simply takes a type id and returns the associated specification. The registry also contains a list of processes who have registered each type, and a counter which tracks the number of values using the type in the kernel.

Two new operations need to be supported by the kernel: register type and unregister type. These are simple operations. The former takes a type specification, stores it in the registry then returns a type id. The latter takes a type id and disassociates the process making the request from the type. If the type count is zero then the type specification is deleted and any associated cached isomorphisms are removed as well.

The rest of the changes that are required are straightforward adaptations of what was described earlier, such as using a Python dictionary to implement the type registry and type isomorphism cache. The others are minor alterations that are required as the tuples are no longer stored or transmitted with their complete type information. The information must instead be looked up from the registry whenever it is required.
Data Structures

The data structures used by the registered implementation’s kernel are identical to those used in the unregistered implementation. However, rather than a type representation being added to the structures on first use, they are added when registered by a process. Aside from this one difference, the structures are used in the same fashion as in the unregistered systems.

Messages

When a client program creates a new type it sends a register message to the kernel containing the type specification. The format for this message is given below. The return message contains the unique type identifier which is to be used in all future communications involving this type.

```
<message>
  <action>register</action>
  <typeobj name="A">...</typeobj>
</message>
```

Much like the unregistered version the in, rd and out messages are all augmented with type information. In the registered version the values and elements are given a typeid attribute rather than a child typeobj element.

```
<message>
  <action>in</action>
  <tuplespace>xyz</tuplespace>
  <tuple>
    <element>
      <value id="1234" typeid="abc">...</value>
    </element>
    <element typeid="def" />
  </tuple>
</message>
```

If a kernel forwards a message to another kernel containing a type that it has never seen before then it will send a message back to the original kernel requesting the type specification. This message has the following structure.

```
<message>
  <action>type_specification</action>
  <element typeid="abc" />
</message>
```

The original kernel looks this type identifier up and returns the type specification.

```
<typeobj name="A" typeid="abc">...</typeobj>
```

All other message used in the system do not contain type information.
6.8 Evaluation Of Changes

So far in this chapter two different rationales for altering the design of a LINDA kernel to accommodate complex type information have been introduced. Each was described in detail, along with a number of methods for increasing the runtime efficiency of the system.

The main difference between the registered and unregistered methods is that the registered version requires more messages to be sent, but the average size of each message is smaller. Without any optimisations the unregistered version will be extremely slow due to the cost of generating an isomorphism for each comparison between two type representations. With all the proposed optimisations the unregistered system should come close to the performance of the registered system, but the larger message size will have some impact. Just how much of an impact will depend on the size and complexity of the type information being used.

An unregistered kernel can automatically register types based on the values coming into the server. Performing this automatic registration requires tuples to be scanned on entry and exit to the kernel (or tuplespace partition) which can be a costly operation given that it is required for every tuple. In the final optimisation that was presented two levels of caching were used. One level stores the type specifications, at the kernel level (i.e. across all tuplespace partitions), while a secondary level keeps track of which types were stored in which tuplespaces. This will use more memory than a single level cache, though less than no cache at all, but will require tuples to be scanned when entering both the kernel and a tuplespace partition. This double scanning can be optimised to a single scan in some cases, such as when a client is placing a value into a tuplespace, but in cases where a tuple is moved between tuplespaces this kind of optimisation is much harder to perform.

In a registered kernel many of the optimisations proposed for the unregistered version come for free. The per-kernel cache of type specifications is an intrinsic part of the system, and no scanning of tuples to replace specifications by identifiers is required as tuples are moved containing only their unique id. The main optimisation proposed was to pre-calculate type isomorphisms by generating them for all the types registered by processes that share a tuplespace, ignoring the ‘uts’. This may calculate too many isomorphisms as a client could have access to more than one tuplespace and only use a subset of its total registered set of types in each.

6.9 Conclusion

This chapter introduced two methods for extending LINDA to handle complex type information. One method sends complete type information in every message while the other sends more messages but each message is smaller as the complete type specifications are only sent once. Sending more messages makes the kernel slightly more complicated as it has to keep track of the type specifications, but this enables the server to perform several interesting optimisations. These optimisations are
also possible to implement in the unregistered version, but without the knowledge provided by the type registry the optimisations are not as simple.

The type isomorphism algorithm described in chapter 4 was complicated, far more complicated than a retrieval from a hash map therefore it is clear that caching type isomorphisms will lead to a large performance gain. Unfortunately it requires the kernel to perform garbage collection on the isomorphisms, as well as the type specifications. It was described how the kernel can keep track of the relationships between types in order to maintain correct counts of the number of values using those types, and therefore continue to be a persistent system which does not leak memory over time.

In the next two chapters the two methods will be used and their real world performance compared. As both systems have their own benefits and drawbacks the choice of which is best will rest on their real world performance.
Chapter 7

Case Studies

All the pieces of our type matching puzzle are now in place. In previous chapters type representations have been described, as have type isomorphism algorithms and the problems of integrating them into an open coordination system. In this chapter a modified LINDA system is used to demonstrate the additional flexibility given to programmers through the use of compound data types. The following chapter will take a more methodical approach and study the actual performance of the system.

To demonstrate the utility of the system, a collection of medium sized case studies will be detailed. As well as being used to benchmark the changes made in the system these case studies will provide concise examples of how complex types increase the flexibility for a programmer and allow novel solutions to problems that were previously impractical to solve. Each case study will follow a similar format. Firstly the goal of the collection of programs will be described, then how this could be implemented without using complex data types. Finally the implementation using complex types will be described and the two versions compared.

In the first case study compound data types are not used so it is possible to use the same implementation in both the modified and unmodified LINDA systems. The next case study can be implemented both using and not using complex types so the differing implementations are compared. The final two case studies are not imple-mentable without compound types (at least not without significant modifications) so no comparison is done with the unmodified LINDA.

The comparisons in this chapter will be qualitative rather than quantitative and will focus on the code that was written for each implementation of the case study. The code will be compared based on factors such as the ‘naturality’ of the language in which the code was written and the complexity of the requirements made on the programmer.

7.1 Prime Sieve

Prime numbers are a topic that have fascinated mathematicians for centuries. As Euclid demonstrated in around 300 B.C. there are an infinite number of primes, and the calculation of large primes is something that scores of computer scientists have
used to test the speed of their fastest computers. It is fitting then that a prime number sieve is chosen as the first test.

A prime number is simply an integer value, so this case study will not look at the advanced type capabilities that have been proposed. Instead this case study will show that the changes to the system make it no worse than it was before the changes were made, and that the difference in the system’s speed does not seriously affect the performance of the application.

Although there are many different ways of determining whether a number is prime or not the approach used in this case study is the simplest method for determining all the prime numbers below a certain \( n \). A number is tested for primality by dividing it by all primes that are less than it. If it is exactly divisible by any of them then it is not prime. Any non-prime numbers less than it will also be divisible by a number that is prime and so can safely be ignored.

The architecture used in this study is single master with many clients. The master creates a list of all numbers from three to \( n \) and then waits for the clients to produce a list of primes. It also creates the first process in the prime sieve chain by specifying that 2 is a prime number. The clients are organised as a collection of processes. Each collection has one process that watches for new prime numbers, and the rest are processes where one process represents a single prime number and each prime number is represented by one process. The processes examine each number coming from the previous prime process in turn, and divides it by the prime it represents. If it is exactly divisible then the number is discarded, if it is not then it is passed onto the next prime. When a number reaches the end of the chain a process is created for that number, and it is marked as a prime.

In this architecture two tuplespaces are used, one for storing the list of primes and another for the clients to work in. The clients’ workspace contains all the numbers that have yet to have their primality decided, and those primes that are awaiting being turned into a process.

The numbers that are currently being processed are represented by a 3-tuple containing the number itself, the prime they are to be tested against next and a counter. The prime sieve only works if numbers are processed in ascending numerical order and LINDA’s nondeterministic nature means this is impossible to guarantee unless a counter is used to order them. Each prime process counts the numbers it passes on to the next highest prime, sending this number as the third element in the tuple. When a prime process is created it knows the first number it is to look at will have zero as the counter, and then increase sequentially from that. This allows the process to force the tuplespace to return the numbers in order.

As stated the client processes are grouped together. Each collection of processes has one that watches the workspace for a new prime number. If it retrieves the prime number then it forks a new process and continues watching. On average the created processes should be evenly spread between all process groups. It is expected that a different process group should exist on each machine in the network, although
#!/usr/bin/python

import sys
import linda

linda.connect()

prime_ts = linda.TupleSpace()
linda.uts._out(("prime_ts", prime_ts))

work_ts = linda.TupleSpace()
for i in range(int(sys.argv[1])):
    linda.uts._out(("work_ts", work_ts))

work_ts._out((2, 0, 0))

for i in range(3, 1000):
    work_ts._out((i, 2, i-3))

while True:
    tup = prime_ts._in((int, ))
    if tup is None:
        break
    print tup[0],

Figure 7.1: The master process.

several process groups could coexist peacefully on a single machine.

Finally, there is some additional code that enables the clients to recognise when the system has reached the end of its run and to shutdown. This is performed using Linda’s deadlock detection [44] and garbage collection to tell each thread to quit in turn and to then clear up the remaining tuplespaces.

7.1.1 Implementation

The first piece of code is given in figure 7.1 and is run once to populate the list of numbers to be examined, create the first prime process and to produce the list of primes at the end of the run. The code connects to the Linda network and creates a new tuplespace. It then adds the tuple (2, 0, 0) to the working tuplespace. This indicates that the number 2 is a prime. The process then places each number between 3 and 100,000 into the uts, specifying that each is to be checked next by the prime process representing 2. Finally the process watches the prime_ts tuplespace for prime numbers.

The code in figure 7.2 represents the controller of the groups of prime processes. It simply watches for numbers that have been found to be prime (indicated by the 0 in the second element of the tuple). Once found it spawns a new process (the “prime_sieve.py” script).
#!/usr/bin/python
import os
import sys
import linda

linda.connect()

_, work_ts = linda.uts._rd(("work_ts", linda.TupleSpace))

while True:
    # Get a new prime
    tup = work_ts._inp((int, 0, int))
    if tup is None:
        break
    # Spawn a sieve process
    if os.fork() == 0:
        os.execlp("python", "python", "prime_sieve.py", str(tup[0]))

Figure 7.2: The slave control process.

This final piece of code is run once for each prime number that is found. The
prime number is passed as a command line parameter from the code in figure 7.3.
The process watches the working tuplespace for a number that it has to check. The
process checks if the number divides exactly by the prime and discards the number,
marks it has a prime or passes it on to the next prime as appropriate.

7.1.2 Conclusion

The fact that the same code can be run on both the modified and unmodified
implementations of LINDA helps to show that the changes proposed in this thesis
do not make LINDA necessarily more complicated. The same code can be run on
both modified and unmodified LINDA systems meaning the changes are ‘backwards
compatible’.

The main function of this case study is to allow the traditional implementation
of LINDA to be compared against the two modified versions in a real world situation.
The results of this comparison are given in chapter 8.

7.2 Sudoku Solver

Sudoku is a simple yet addictive logic puzzle first created by the retired architect
Howard Garns and published in 1979. The puzzle became hugely popular in mid-
2005 when puzzles were published in many daily newspapers. Puzzles range in
difficulty from easy to the fiendishly difficult. The fiendish puzzles can prove very
tricky for a human to solve, although a computer will typically have no such trouble.

A Sudoku puzzle consists of a 9 by 9 grid of numbers and blank squares. The
import sys
import linda

linda.connect()

prime = int(sys.argv[1])

# Let the master know we've found a prime
_, prime_ts = linda.uts._rd(\"prime_ts\", linda.TupleSpace)
prime_ts._out((prime, ))

_, work_ts = linda.uts._rd(\"work_ts\", linda.TupleSpace)

next_prime = 0

count = 0
outcount = 0
while True:
    tup = work_ts._inp((int, prime, count))
    count += 1
    if tup is None:
        break
    # check if the number being tested is divisible by us
    if tup[0] % prime == 0:
        continue
    elif next_prime == 0:
        # this is the first possible prime we've found
        work_ts._out((tup[0], 0, 0))
        linda.uts._out(\"work_ts\", work_ts)
        next_prime = tup[0]
    else:
        work_ts._out((tup[0], next_prime, outcount))
        outcount += 1
print \"done\"

Figure 7.3: The prime sieve process.
struct SudokuSquare {
    enum {
        BLANK,
        FILLED
    } type;
    int value;
}

struct SudokuGrid {
    struct SudokuSquare[9][9];
}

Figure 7.4: A Sudoku Grid in C

class SudokuRow:
    def __init__(self):
        self.rows = [None for _ in range(9)]

class SudokuGrid:
    def __init__(self):
        self.grid = [SudokuRow() for _ in range(9)]

Figure 7.5: A Sudoku Grid in Python

object is to fill in each blank with a number from 1 to 9 such that every row, column and 3 by 3 square features each number once and only once. There are a number of additional rules that designer should follow if it is to be a ‘proper’ Sudoku puzzle. There should be no more than 32 given numbers, the given numbers should be symmetrical and the puzzle should be solvable without resorting to guess work.

A Sudoku grid can be represented quite naturally as a two dimensional 9 by 9 array of a sum type containing a null value or an integer. Figures 7.4 and 7.5 show a type representing a Sudoku grid implemented in C and Python respectively. These types are both representations of the same abstract ‘Sudoku grid type’.

As has been described earlier, before a concrete type can be used it must be converted into an abstract representation so it can be shared between the client and the LINDA kernel. The C implementation can be naturally represented as ‘sudoku-c: (sudoku-sq * 9) * 9; sudoku-sq: Null + int;’\(^1\). The Python version translates easily to ‘sudoku-py: row * 9; row: (Nil + Int) * 9;’. These two types, following quick inspection, are clearly equivalent and indeed the type isomorphism algorithm will determine this to be the case. Of course the programmer need not work this out for themselves, but by substituting sudoku-sq and row where they are used the types end up as ‘sudoku-c: ((Null + Int) * 9) * 9;’ and ‘sudoku-py: ((Null + Int) * 9) * 9;’ which are equivalent.

\(^1\)The 9 in the type specification is a short hand for a nine element product type, with each element containing the same type. The type t :: int * 3; is equivalent to t :: int * int * int. Without this short hand specifying a 9 by 9 grid is extremely cumbersome.
In this case study we are imagining the solving of Sudoku to be a computationally expensive process thus the processes that solve the puzzles are written in C for performance, while the controlling process is written in Python for ease of use. The processes will be divided into groups, the C process will watch a tuplespace for a partially solved puzzle while the Python process will perform the interaction with the user, i.e. retrieving the puzzle to be solved and displaying the solution.

The client process works by taking a Sudoku puzzle out of the tuplespace, it then scans the grid starting in the top left hand corner and moving down then right through the grid. At each square it checks if there is a value there, if so then it moves on to the next square. If the square is empty then it goes through each number 1 to 9 and checks if placing it in the square would violate the three key rules of Sudoku (only one of each number in each row, column and 3 by 3 square). If the number is valid then it puts the grid, with that number placed, into the tuplespace. If it is not valid then it checks the next number. Once all numbers have been checked it discards the grid and waits to retrieve another grid from the tuplespace.

The tuples that will be used to contain the Sudoku grids contain one other value in addition to the grid. This number is the number of squares that remain to be filled in and is used so the master process can watch exclusively for completed grids. The master sets this value to the number of squares that are left when it creates the grid, then each client decrements this value by one when it fills in a square.

The code for implementing this is given in section D.2.2.

7.2.1 Without Compound Types

In order to perform a comparison between the modified and unmodified LINDA systems it is necessary to rewrite this case study so that it does not use compound data-types. To represent a Sudoku grid without using compound types there are two choices, either a value representing a grid is encoded as string and both programs must then be able to encode and decode the same format, or an 81 element tuple can be used. An 81 element tuple is large and unwieldy to use but it does avoid having to write code to parse a string into a grid.

To provide an accurate comparison the method used to calculate the solution is identical in both versions, the only difference is in the tuples that are passed around. As the 81 elements can only be integers a Null value cannot be used to indicate a blank, therefore a blank is indicated by -1. The number of blanks is also indicated in the tuple, and it is arbitrarily placed as the first element of the full 82 element tuple, though it could be placed anywhere in the 82 elements as long as both processes looked for it in the same place.

The code for implementing this is given in section D.2.1.

7.2.2 Comparisons

Figures 7.6 and 7.7 show the two partial versions of the master processes (the full code can be found in sections D.2.1 and D.2.2). Although the serialisation functions
def master():
    grid = loadFile(sys.argv[1])

    print "Solving...
    print gridString(grid)

    empty = sum([len([x for x in row if x is None])
                 for row in grid])

    if not linda.connect():
        print "Please start the Linda server first."
        return

    start = time.time()
    linda.uts._out((empty, ) + gridToTup(grid))

    tup = linda.uts._in((0, ) + (int, ) * (9*9))

    print "Solved Grid in %f seconds.\n" % (time.time() - start)
    print gridString(tupToGrid(tup[1:]))

Figure 7.6: Traditional version of the Master process

def master():
    grid = loadFile(sys.argv[1])

    print "Solving...
    print gridString(grid)

    empty = sum([len([x for x in row if x is None])
                 for row in grid.grid])

    start = time.time()
    linda.uts._out((empty, grid))

    tup = linda.uts._in((0, gridtype))

    assert tup[0] == 0, tup[0]

    print "Solved Grid in %f seconds.\n" % (time.time() - start)
    print gridString(tup[1])

Figure 7.7: Typed version of the Master process
in the typed version are complicated they are significantly easier to understand than
the functions which convert from the grid type into a tuple in the traditional version.
As the typed version is treating a grid as a single value in the tuple the serialisation
function is not concerned with the number of free spaces in the grid. The traditional
version has to convert the grid type into an 81 element tuple, and attach the number
of free spaces.

7.2.3 Conclusion

This case study shows that not only do the extensions to LINDA proposed in this
thesis allow the programmer to express the values they wish to use in a more precise
manner, but that this reduces the amount of code they have to write as the conver-
sion between values in their program and those in LINDA occurs automatically.

In this section the difference in speed of the two methods of programming and
the three implementations has not been considered. This case study will be used as
the basis for studying speed of the various implementations in section 8.8.

7.3 Sudoku Generator

This case study is the reverse of the previous system. Instead of solving a problem
the aim is to generate a puzzle of a certain difficulty. The important rules relating
to the creation of Sudoku problems are as follows.

- There should be no more than 32 given numbers.
- The given numbers should form a symmetrical pattern.
- The puzzle should be solvable without resorting to guess work.
- There should only be one solution.

There are many methods of creating a Sudoku puzzle, the method used here
is slow but will produce good puzzles that fit all of these criteria and are of the
specified difficulty. The system is divided into three groups of processes, the master
that controls the system, the solver slaves and the generator slaves. The generator
slaves take a partially generated puzzle, adds a number to the puzzle then uses the
solver slaves to see if it has a single solution. As part of checking that it has a single
solution a function is run over the puzzle to test its difficulty. If the difficulty is just
right then the puzzle is returned, but if it is not it is placed back into the tuplespace
to be altered and tested again.

The solver slaves work in a different fashion to those in the previous case study,
instead these slaves perform all the solving in the same process. Rather than being
concerned with the actual solution they are concerned with whether the Sudoku has
only one solution and its difficulty level.

The decision on whether a puzzle is of the correct difficulty level is, in itself,
a difficult problem. Rather than have each process have the various methods for
calculating difficulty programmed into them, a function will be placed into the tuplespace by the master process. This function will take a Sudoku grid and return a boolean value, indicating whether it is of the required difficulty or not. By passing the function from the master to the various client processes the function can be easily changed depending on the requirements of the user.

### 7.3.1 Testing Grids

Determining the difficulty of a Sudoku grid is itself a difficult and subjective problem. How to calculate the difficulty is essentially a side issue to the one being investigated in this case study. The program developed will not work without a function which takes a grid and determines if it is of the correct difficulty, so a simplistic function is used and given in section D.3.4. How the difficulty function determines whether a Sudoku is of the correct difficulty is irrelevant for this case study, the important aspect is the type of the function which is given in figure 7.8.

The key function in the difficulty testing program is `difficulty`. It takes a Sudoku grid and returns a boolean to indicate whether the grid is acceptable. It performs this test by ensuring that the grid has only 32 elements, that the given numbers are symmetrical and that no guess work is required in solving it.

The process that evaluates the function is given in section D.3.3. Although the client code is quite long and involved, much of the code is simple boilerplate to handle memory management (the `Linda_addReference` and `Linda_delReference` calls) and to create the tuples. The client process repeatedly reads a grid from the main tuplespace and applies the retrieved function to it. If the function returns true then the grid is marked as a successful Sudoku grid, otherwise it is discarded. This leads to extremely simple code as most of the work is performed by the function given to the process. The key line is \( r = \text{Linda_apply}(\text{difficulty}\_\text{func, args}) \), where `args` is a tuple containing the Sudoku grid. This line calls the function to determine whether the Sudoku grid is valid. Despite the simplicity of this line the function call could perform any test on the grid as the function is retrieved, at runtime, from the `uts`.

### 7.3.2 Generating Grids

To generate grids an empty grid is taken, then a single number is added to the grid. A number is repeatedly added to the grid, which is tested after each one. The only concern of the generating process is that each number that is added does not produce an invalid puzzle. This is different from being concerned that the puzzle is
valid, as that would involve checking for symmetry and for a single solution, and is a much simpler problem.

The code for this function is given in section D.3.2.

7.3.3 Types

Each of the three processes that form this case study use a different type to represent a Sudoku grid. The types are all isomorphic but the LINDA kernel ensures that values from any process can be used by any other process. The function that is used to determine the difficulty of the grid is defined using the types of one process, but evaluated using the types of another. The kernel generated wrapper function performs the conversion between values of the different types transparently when the function is called.

The three programs that form this case study each represent a Sudoku grid differently.

- grid :: ((int + Nil) * 9) * 9;
- grid :: row * 9; row :: square * 9; square :: Nil + int;
- grid :: ((Nil + int) * 9) * 9;

Despite each program representing types differently the LINDA kernel detects that all three are isomorphic and automatically converts values between the types. This is perhaps the clearest example in these case studies of how different programs can be developed independently and how they are not required to use the identical type specifications, instead they can use types that are tailored to their exact requirements.

7.3.4 Non Isomorphic Types

Consider a Sudoku generator that is implemented without using the compound type features available in the modified LINDA system. If it encodes a grid as a string, using a type defined as grid :: string; it will be able to work alongside the version given above as grid :: string; is not isomorphic to any of the other grid types.

7.3.5 Conclusion

In this section a much larger example than the ones previously shown has been introduced. This example involves three different programs written in two different languages using three different representations for the grid type. Although each process has a specific function, and they collaborate together to generate the grids, this is all that needs to be decided prior to writing any code. The programmer is freed from worrying about how the functions will communicate or transfer the grids: it all happens automatically and is type safe.
7.4 Triangles

In this section a system will be introduced which calculates the missing side of a right angle triangle. The basic structure will follow the same master/slave pattern that was used in previous examples. The slave process will read in a function before each triangle, and will execute the function on the triangle and place the resulting value back into the tuplespace.

It is assumed that the first of the three numbers that defines the triangle is the horizontal side, the second is the vertical and the third is the hypotenuse.

7.4.1 Slave

The slave process given in figure 7.9 defines three types, one for a partially complete triangle, one for a complete triangle and one for the function that will be used to fill in the missing side. The process repeatedly reads the function from the tuplespace and uses it to fill in one triangle.

```python
def slave():
    partial_triangle = linda.Type("partial :: (float * float * nil) \+
        (float * nil * float) + (nil * float * float)")
    full_triangle = linda.Type("full :: float * float * float")
    func_type = linda.Type("func :: partial -> full;")

    while True:
        (func, ) = linda.uts._rd((func_type, ))
        (triangle, ) = linda.uts._in((partial_triangle, ))

        triangle = func(triangle)
        linda.uts._out(triangle)

Figure 7.9: The Slave Process

We will not consider the master process here, but will assume that there is a process running which outputs a series of incomplete triangles and reads in the completed versions from the uts. We will also assume that the master process does not control the function used to complete the triangles. The program in figure 7.10 outputs a function which uses Pythagoras’ theorem to do that.

In combination with a master process these two programs form a complete system which matches the specification given at the beginning of this section. It is important to note that the entire system works in a single tuplespace, and that the partial triangle and complete triangle types are not isomorphic so the slave process does not retrieve a complete triangle when it is requesting a partially complete one.

7.4.2 An Additional Process

The next process (figure 7.11) is another slave process which will work alongside the one given above, but it has trigonometry hardcoded as its method of completing the
The function `complete` is defined as follows:

```haskell
func = linda.Function(""
complete :: (float * float * nil) + (float * nil * float) + (nil * float * float) -> (float * float * float);
complete t = if t[0] == Nil then
  (sqrt(t[2]*t[2] - t[1]*t[1]), t[1], t[2])
else if t[1] == Nil then
  (t[0], sqrt(t[2]*t[2] - t[0]*t[0]), t[2])
else
  (t[0], t[1], sqrt(t[0]*t[0] + t[1]*t[1]))
"")
```

This new function will work alongside the functions given before and will complete the triangles using a different method, in a different language.

**7.4.3 Conclusion**

This example shows three interesting aspects of the system proposed in this thesis. Firstly, it shows that similar but distinct types are not mixed. Secondly, it shows how functional values can be used to simplify programs by making them configurable at runtime. And thirdly, it shows that the system retains its openness by introducing a new program into the system, which is written in a different language.

**7.5 Implementing **eval**

In this case study, a primitive from LINDA’s earliest days, but one that has fallen out of favour, is reintroduced. The changes proposed in this thesis mean it can be implemented without modifications to the LINDA kernel, which were previously required.

**eval** is used to create an ‘active’ tuple. An active tuple differs from a passive tuple in that one or more of its elements are processes which are executing. When one of the processes finishes executing the process returns a value, which replaces the process in the tuple. Once all of the processes have finished executing the tuple becomes passive.

In the original implementation of LINDA, processes were contained in elements of a tuple rather than one process creating one complete tuple. This means that the LINDA kernel is free to send each element of the tuple to a different machine to be executed, thereby spreading the load across multiple servers. An active tuple cannot be matched and removed by a process before the evaluation of all the processes has finished. Therefore, the fact that this implementation will use a single
process to create a complete tuple is not significant as the two implementations are indistinguishable from each other by other client processes.

### 7.5.1 Implementation

To implement the `eval` primitive a watcher process is required to monitor the `uts` for active tuples. This monitoring takes the form of a simple in operation. The template for an active tuple is `(some unique key, Nil → Nil)` where `some unique key` is a value that is used to indicate that the tuple was placed there through a call to `eval`. It is not unique for every call to `eval`, but instead should be unique in that it is not used in any tuples other than those created through `eval`. `Nil → Nil` is the function which will create the final tuple. When the MINIMAL language was defined it was specified such that functions did not have side-effects, with a special exception made for the LINDA primitives.

The monitoring process needs to be run on every node in the LINDA network to ensure that active tuples are spread throughout the network, and not grouped on one node. It will not be considered how to make sure this happens automatically, instead it will be assumed that when starting a LINDA kernel the monitoring process will also be started.

The client processes need a function to be implemented which can take a function and place it in a tuple with the unique key. This function needs to be implemented in each client language, but it is extremely simple and will follow the same pattern for each language. Figure 7.12 shows this function implemented in C.

The function `eval` takes a single parameter, the function which will create the tuple, and places the function into `uts`, with the correct identifier.

The monitor code in figure 7.13 is written in C, but as MINIMAL functions can be transferred from any language to any language, this is not a requirement. The process simply creates a template to match a tuple which was outed by the `eval` primitive. The process then repeatedly matches on the template and evaluates the function which was retrieved.

The function is evaluated in the same thread that retrieved it. This means that where `n` monitoring processes are running only `n` functions can be executed concurrently. Extending this to execute an arbitrary number of functions concurrently by evaluating each function in a new process or thread would be straightforward.

Figure 7.14 shows a simple function which when evaluated puts a single tuple, (1) into `uts`. Figure 7.15 shows a slightly more complex example, in which a single function expands to ten tuples.

### 7.5.2 Conclusion

This section has shown that given support for functions in LINDA it is easy to implement the `eval` primitive without explicit support for it from the kernel. Although this implementation of `eval` is not as cleanly integrated with LINDA as the original
versions, which were part of the kernel, it is still a useful and easy to use tool for the LINDA programmer.

7.6 Conclusion

From the case studies presented in this chapter it is clear that there is certain level of complexity required to use complex types. The requirement of writing code to serialise values is unfortunate, but whether explicit, as it is here, or generated automatically through the use of a complex preprocessor, it is required. The Sudoku solver is an excellent example of how the use of complex types simplifies matters for the programmer as the serialisation code is specific to one type, whereas being forced to decompose the grid type into atomic types involves combining a large number of integers with the integer representing the number of free spaces.

The Sudoku generator and eval case studies show how functions enable the programmer to create a single, simple program which is then customised and controlled at runtime by a master process. The process implementing the eval primitive is completely abstract, and it depends solely on the function which is retrieved.

Each case study uses a small set of types that does not change as the system runs, except in the case of the triangle case study. In that example, a new program was introduced as the system was running which registered a few new types. However, after that the types in the system stayed constant. This results in the set of comparisons between types being consistent and small, perfect for the type isomorphism cache that was described previously. While it is possible to generate a large number of types as the system runs this is unlikely to occur and the behaviour of the system is more likely to be like that of the case studies.

As there is no difference from the programmer’s perspective between the registered and unregistered implementations this chapter has treated them as the same. However, they will have different performance characteristics, and will be compared to each other and a traditional LINDA system in the next chapter.
int main(int argc, char* argv[]) {
    Linda_connect();

    LindaValue partial = Linda_type("partial :: (float * float * nil) \\
    + (float * nil * float) + (nil * float * float)");
    LindaValue full = Linda_type("full_triangle = linda.Type( \\
    "full :: float * float * float"));

    LindaValue match = Linda_tuple(1);
    Linda_addReference(partial);
    Linda_setTuple(match, 0, partial);
    while(1) {
        LindaValue t = Linda_in(Linda_uts, match);

        LindaValue triangle = Linda_getTuple(t, 0);
        if(Linda_isNil(Linda_getProduct(triangle, 0))) {
            float op = Linda_getFloat(Linda_getProduct(triangle, 1));
            float hyp = Linda_getFloat(Linda_getProduct(triangle, 2));
            float angle = asinf(op / hyp);
            Linda_setProduct(triangle, 0, Linda_float(cosf(angle) * hyp));
        } else if(Linda_isNil(Linda_getProduct(triangle, 1))) {
            float adj = Linda_getFloat(Linda_getProduct(triangle, 0));
            float hyp = Linda_getFloat(Linda_getProduct(triangle, 2));
            float angle = acosf(adj / hyp);
            Linda_setProduct(triangle, 1, Linda_float(sinf(angle) * hyp));
        } else if(Linda_isNil(Linda_getProduct(triangle, 2))) {
            float adj = Linda_getFloat(Linda_getProduct(triangle, 0));
            float op = Linda_getFloat(Linda_getProduct(triangle, 1));
            float angle = atanf(op / adj);
            Linda_setProduct(triangle, 2, Linda_float((1.0/sinf(angle)) \\
                * op));
        }
    }
    Linda_setType(triangle, full);

    // triangle refers to the triangle stored in t,
    // so we don’t need to update t
    Linda_out(Linda_uts, t);
}
}

Figure 7.11: Using C to complete the triangle.
void eval(LindaValue func) {
    LindaValue t = Linda_tuple(2);
    Linda_setTuple(t, 0, Linda_string("__eval__"));
    Linda_addReference(func);
    Linda_setTuple(t, 1, func);

    Linda_out(Linda_uts, t);

    Linda_delReference(t);
}

#include "linda.h"

void main(int argc, char* argv[]) {
    Linda_connect();

    LindaValue template = Linda_tuple(2);
    Linda_setTuple(template, 0, Linda_string("__eval__"));
    Linda_setTuple(template, 1, Linda_type("func :: Nil -> Nil;"));

    while(1) {
        LindaValue tup = Linda_in(template);

        Linda_callFunction(Linda_getTuple(tup, 1), Linda_nil);

        Linda_delReference(tup);
    }
}

func = linda.Function(""
    example :: Nil -> Nil;
    example = out(uts, (1, ));
    ""
)

linda.eval(func)

Figure 7.12: An implementation of eval in C.

Figure 7.13: The eval monitor code in C.

Figure 7.14: A simple example of the use of eval in Python
func = Linda.Function(""
loop :: Nil -> Nil;
loop x = if x == 0 then out(uts, (x, )) else
  let _ = out(uts, (x, ));
  loop(x-1);
endif;

example :: Nil -> Nil;
example = loop(10);
"")
eval(func)

Figure 7.15: Creating multiple tuples from a single eval in Python
Chapter 8

Results

In this chapter an implementation of the techniques proposed in this thesis will be tested to measure various performance characteristics, in a variety of situations. The implementation has been created as an extension to PyLinda which originally provided a traditional version of LINDA (i.e. one without support for compound types). The extension provides both of the proposed registered and unregistered versions of typed LINDA, selectable at compile time. This allows both methods for handling type information to be compared, and for the traditional untyped implementation to provide a benchmark against which the extensions can be measured.

As the typed versions of LINDA have been created as an extension to an untyped implementation they share the same core code. For tests where the typed extensions are compared against the untyped version the only differences are in the changes required to support types. Although PyLinda is not a fault-tolerant, highly scalable implementation of the LINDA model its deficiencies (and its strengths) are shared between all three versions being measured.

Ideally there would be an existing corpus of work for LINDA and LINDA-like systems that could be used as the basis for the comparisons in this chapter, unfortunately such a thing does not exist. The tests described focus on testing the LINDA system as a whole and do not split the system up into components. By carefully selecting the tests the performance of the components can be measured without the inconvenience of using the LINDA kernel in an unconventional manner.

The three systems being compared are the traditional, registered and unregistered versions. As the traditional version does not support compound types it sends much less type information and has the simplest type matching algorithm, so it should be the quickest. The unregistered version does support compound types and it sends the complete type information with every value. This will use much more network bandwidth than the traditional version. This, combined with the process of comparing types to the types already seen by the kernel, will mean it is considerably slower. The registered version supports compound types and sends a unique id instead of the complete type information. This results in an additional message being sent for each type registered. This is a small overhead compared with the unregistered version, and while the registered version should be expected
to be marginally slower than the traditional version it will be much faster than the
unregistered implementation.

The tests will compare the amount of time taken to perform the tasks, the largest
amount of memory consumed while doing so and the amount of network traffic
involved. These three metrics cover all the key points of a running LINDA system.
The measurement of time will demonstrate any change in the overall performance
of the system, including any change in the amount of processing the client or LINDA
kernel has to do as a result of the proposed extensions. The amount of memory
used will only have a small effect on the performance of the system, unless it pushes
the node into using virtual memory - which would have a catastrophic effect. The
amount of memory used has an affect on the system as more memory is used per
item of data stored the fewer items the kernel can store before using virtual memory.
Network traffic measures the amount of data sent from clients to kernels and between
kernels. Network bandwidth is a finite resource and when a network connection is
saturated, data is delayed, causing a slow down in the system.

In order to perform these tests the system was run on a network of virtualised
servers which form part of Amazon’s Elastic Compute Cloud (EC2) service\(^1\). The
network is made up of a number of nodes, each with the equivalent of a 1.7Ghz x86
processor, 1.75GB of RAM, 160GB of disk space and 250MB/s of network band-
width. All machines run a Linux 2.6 series kernel patched with the Xen modifications
required for it to be run as a virtualised server. All the software run is compiled
with GCC 4.1, and the Python parts of the system are run on Python 2.4.1. The
servers themselves run as few processes as possible to reduce any external factors
which would effect the timings. The only processes running besides those involved
in the test are an SSH daemon and a small Bash shell script which controls the
tests. Both these processes are idle during the test runs and will not affect any of
the measurements taken.

Details on how to retrieve and run the software used for these tests can be found
in appendix C. As the source code used for these tests is open source, and the
hardware used to run them is publicly available the results presented in this chapter
should be repeatable by anyone.

\subsection{Testing Environment}

Some tests will be conducted on just one machine, for these tests the network band-
width available (through Unix Domain sockets) is so large that it has a negligible
effect on the tests when compared to inter-machine communications. Other tests
will involve more than one machine communicating and will therefore be affected
by the current network conditions. As the tests are not being run on an exclusive
network (there are other users who can perform network operations at any moment)

\footnote{For more details on the advantages and pitfalls of using virtualised servers as a basis for the
testing of distributed applications see appendix A}
the first test will simply measure the latency of the network to judge how inter-kernel communication will be affected by environmental factors.

In a system where the current process can fit entirely into memory, the network is likely to be the biggest bottleneck. Although technology is increasing the speed of network communications it is not unusual to see latencies of tens or even hundreds of milliseconds between computers on the Internet. When communicating over a Local Area Network (LAN) the latencies are much lower, typically around 1ms. However, this is still a significant delay with respect to the processing speed of an average PC.

Latency is the time in which it takes a message to travel from one machine to another. Bandwidth is the rate at which data can be transferred across a network. If a van filled with DVDs were to drive from Lands End to John O’Groats\(^2\) then it would have a high latency (which is bad for real-time communication) but also a high bandwidth (which is good for transferring data). Latency is partly caused by the physical properties of the communication medium but also by the processing that must be performed at each end to encode and decode the message. The available bandwidth is limited by the rate at which packets can be transmitted, and their size. Each packet is divided into two sections, a message header and a body. For a TCP/IP packet this message header contains details such as the message length, a checksum and the source and destination IP addresses. From an application’s perspective, space taken up by the header is wasted bandwidth.

In this first test a series of clusters containing twenty nodes will be started a various times across various days. Each node pings each of the other nineteen nodes in turn. Each ping consists of one hundred Internet Control Message Protocol (ICMP) packets, sent at one second intervals. The time taken for the node to receive an ICMP echo response is averaged across all one hundred round-trips. Each node generates nineteen data-points, giving 380 results per run and 1,900 in total.

The vast majority of runs measured the latency as extremely low. The mean latency was only 7.08ms, with a median of just 3ms and a standard deviation of 20.6ms. In 80% of the cases the latency was below 5ms, with 95% of all tests showing that the latency was under 15ms. In a small number of cases (1%) the tests measured the latency as over 100ms. Given that network latency is only one factor that effects the performance of the system the rare high latency events will not have a significant impact on the system.

8.2 Measureables

In order to give a correct view of the performance of the various implementations it is important that the measurements are taken correctly. The three sections below take each of the measured items in turn and discuss how to measure them accurately.

\(^2\)A journey of over 800 miles taking at least 20 hours
• **Time**

The time taken by a process to run can be divided up in two ways. Firstly there is real time, which is the amount of time that passes in the outside world between the process starting and finishing. Secondly there is CPU time. CPU time is the amount of time that the process has had in control of the CPU. Real time includes time that the process spent waiting for communication from the network, access to the disk or was otherwise blocked. Real time gives a good indication of the overall performance for a system. It is extremely easy to measure to sub-second accuracy and so this is the measure that will be used. CPU time ignores the time spent blocked by a process and measures how long it spent actually performing work. As a LINDA system is made up of multiple processes running across multiple machines it is difficult to calculate CPU time accurately and therefore this measure will not be used.

• **Memory**

The additional type information that needs to be sent with data will inevitably cause the memory usage of the LINDA kernel to increase. The extent to which it is increased will depend on the size of the type and the number of tuples stored. This section will measure the memory usage of a single LINDA kernel with various numbers of tuples being stored with a variety of types as elements.

Measuring the memory usage of a process on a modern operating system is not an easy problem. Virtual memory means that memory that is in use may not be resident and so may not show up on a usage chart. Memory may also be shared with other processes and so may be counted twice. A recent addition to the Linux kernel is smaps\(^3\). This provides a simple file as part of the /proc filesystem for each process which contains a detailed outline of its memory usage. The memory usage will be measured by totalling the values in this file.

• **Network Traffic**

There are two forms of networking used by the implementation being tested. While they appear the same to the programmer they have very different characteristics.

- **TCP/IP communication**

TCP/IP is a networking protocol generally used when connecting processes on separate machines, but it can also be used between processes on the same machine. It is used by the vast majority of private networks and on the Internet. It provides reliable, in-order messaging which allows client applications to build protocols on top of it without worrying about missing or wrongly ordered packets. This reliability comes at a price as packets may need to be re-sent or delayed to let previous packets

\(^3\)http://bmaurer.blogspot.com/2006/03/memory-usage-with-smaps.html
TCP/IP divides messages that are longer than the Maximum Transmission Unit (MTU) into multiple packets. When communication is performed between two processes on the same machine the MTU is set to the maximum possible value, 16436 bytes.

- **Unix Domain Sockets**

  Unix Domain Sockets are a standard implementation of named pipes that are available on Linux and many other Unix variants. The endpoint of a Unix Domain Socket appears as a file in the file system, which can be opened and written to. A server process creates the socket and listens for incoming connections, exactly the same as if it were a TCP/IP socket. When a client wishes to connect to the server it creates a socket using the filename of the socket object. Once this has been done both processes may write to and read from the socket regardless of the socket’s type.

  As Unix domain sockets are only for communication between local processes the messages do not need to be encapsulated in a packet as is the case with TCP/IP. This reduces the amount of code that processes the message, decreasing the latency and increasing the throughput of the socket. As the message never leaves the local machine’s memory it does not need to be divided into separate packets no matter how large it is.

  Although TCP/IP is designed for communication between machines it can be used for communication between processes on the same machine. Although Unix Domain sockets are faster (by approximately 10% on Linux) as they used shared memory they are not always available. Windows, for example, does not implement Unix Domain Sockets. Unix Domain Sockets require write access to a file system to create the socket ‘file’, if write access is not available then they cannot be used.

### 8.3 Outputting Tuples

One of the important aspects of a LINDA-like system is the performance of outputting a tuple. If placing a tuple into a tuplespace takes a long time then, clearly, other processes will not be able to retrieve it in a timely manner. In this test the three systems were timed while outputting ten thousand copies of the same tuple into the tuples. The tuple simply consists of a number of elements, each containing the integer value 1. The tuple is increased in size from zero to forty eight elements, in steps of three. This will show how the system performs with very small to moderately large tuples. Beyond forty eight elements the tests start to take too long to run to be practical, and the values tested below that level give a good indication as to the performance of the system. While the number of tuples could be reduced to enable
linda.connect()

tup = (1, ) * size

start = time.time()
for i in range(10000):
    linda.uts._out(tup)
end = time.time()

Figure 8.1: Code For The Out Tests

![Graph showing time taken to output 10,000 tuples]

Figure 8.2: Time Taken To Output 10,000 Tuples

longer tuples to be used a high number must be used to give accurate timing results for small tuples. The results are very consistent and it is expected that the trends will continue for larger tuples.

Before each test for a particular tuple size is run the LINDA kernel is restarted and a new client program is started. Once the client program is connected it outs a tuple 10,000 times. The tuple is constructed prior to the loop so the only code that is executed in the timed loop is directly related to the process of placing a tuple into a tuplespace. The code listing for this test is given in figure 8.1. As with all the tests the time it takes to run for each tuple size is measured five times with the results averaged to give the final values.

Figure 8.2 shows the results of this test. It is immediately clear that the traditional version of LINDA is quickest, and that the time taken scales linearly with the size of the tuple. Both the registered and unregistered versions take longer to run, by 25% and 86% on average respectively. The difference between the registered
t = linda.Type("t :: int * %i;" % (int(sys.argv[1]), ))

val = linda.Value((1, ) * size, t)

tup = (val, )

start = time.time()
for i in range(10000):
    linda.uts._out(tup)
end = time.time()

Figure 8.3: Code For The Compound Out Tests

and unregistered versions is due to the automatic registration of types (see section 6.3), which is done for each tuple sent. The registered version does not need to do this, and is slower than the traditional version because of the extra information associated with each value with a type that must be sent.

The cost of automatically registering types in the unregistered version is prohibitively high. An 86% increase in the time taken to run the simplest test available is too big to justify the simplicity in implementing it. The 25% increase seen in the registered version is significant, but given the extra information that is transmitted it is to be expected.

8.3.1 Complex Types

Outputting tuples containing just the integer value 1 makes no use of the complex types that are the main focus of this thesis. The second test will use the two extensions of LINDA to output 10,000 tuples, but rather than having zero to forty-eight elements they will all contain a single element. This single element will be a value of a product type which has zero to forty-eight elements, each element being the integer value one. Although this uses only one example of how the multi-element tuples can be represented using complex types, it will show the difference in performance between using complex types to more accurately describe the data being transferred, and using longer tuples with simpler types.

The code used for this test is given in figure 8.3 with the results in figure 8.4. The graph shows the line for the traditional version from the previous test for comparison. This line is exactly the line from the graph in figure 8.2. Although the traditional version was run using different code and therefore cannot be directly compared it is useful to consider how the style of programming which is enabled by compound data types affects the performance. In this test the difference between the registered and unregistered version is even more apparent. The time taken to output a tuple containing a compound type increases worse than linearly because the cost of matching the type used when registering increases as the size of the type increases. On average the unregistered version is 166% slower.

The registered version shows an entirely different result as the time taken for it
to run is *better* than the traditional implementation when compound types are used. On average the compound typed version takes 16% less time to execute than the traditional version, while the non-compound type version was 25% slower. This is at first sight counter intuitive, as the compound type version is sending slightly more information (it has to send a product type inside a tuple, and the id of the type of the value) so it could be expected to be slower. Previously it was mentioned that tuples must be scanned to ensure the tuplespace references are correctly handled for garbage collection. When each type is registered it is checked to see if it contains a tuplespace, which in this example they do not, so in this example the value used is not scanned for tuplespace references. In the traditional version every element of the tuple must be checked, while in the compound version there is only ever one element to check. Although for very small tuples the traditional version is faster, as the tuples get larger the number of comparisons that must be made increases and therefore the advantage the registered version has also increases.

### 8.3.2 Memory Usage

Outputting 1,000 identical tuples\(^4\) to a tuplespace is a good comparison for the memory usage of the LINDA kernel. This test will use the code in figure 8.1 but will ignore the time taken to output the tuples and will instead record the peak memory used by the kernel once all 1,000 tuples have been placed into the \texttt{uts}.

\(^4\)The additional work in tracking network and memory usage means that using 10,000 tuples for these tests would take too long to be practical. In these test the values are still accurate even for small tuples as there is less random variation in the memory and network bandwidth used, compared to the time taken.
As is expected the graph in figure 8.5 shows that the traditional version uses a very small amount of memory to store values. Roughly equal in memory usage to the traditional version is the registered version with compound types. This is an interesting result as the registered version without compound types uses 31% more memory for the 48 element test. This is an equivalent to 4KB per tuple stored (for smaller tuples it is less). Storing objects in Python has a large overhead and for the version without compound types the server must create many Python objects. With compound types the value is contained in a single Python object, limiting the overhead. With a different implementation language or strategy this overhead could be avoided. While this is fair representation of the system as it is currently implemented, much of the extra memory usage is not directly caused by the different strategies used.

The two unregistered tests both show a significant memory overhead when compared to the traditional system. The compound type version has just over double the peak memory usage of the traditional version for a 48 element tuple. Further investigation revealed that this peak occurs while registering the type sent with the value. Once the type has been registered the memory usage decreases to a value similar to the registered versions.

8.3.3 Network Usage

PyLinda is designed so that the client and the kernel are separate processes which communicate via a networking stack. As the client will always connect to a kernel running on the same machine they communicate using Unix Domain Sockets rather
than the more expensive TCP/IP stack, if Unix Domain Sockets are available. As Unix Domain Sockets have such high bandwidth compared to TCP/IP the amount of data transferred has little effect on the speed. However, for completeness and for systems where Unix Domain Sockets are not available the amount of data transferred will be measured. The total amount of data transferred from the client to the server was measured while outputting 1,000 tuples. As the server only sends back a simple acknowledgment the bandwidth used in the return messages is the same for each system and was not measured.

As is expected the traditional system has the lowest network usage, which is due to the minimal amount of type information being sent. The two registered tests use an almost identical amount of bandwidth. The version without compound types uses marginally more bandwidth than the one with compound types, this is because each value in the tuple is associated with a type id, while with the compound type this id is only sent with the one value. The two unregistered versions use considerably more bandwidth as they send the complete type information with every message. The compound type information is much larger than the simple integers being used in the non-compound type version, leading to the gap between the two test results.

### 8.4 Retrieving Tuples

After placing a tuple into a tuplespace the usual operation that is performed next is to retrieve the tuple. In this test 10,000 tuples will be placed into the utks, as was the case in the previous test, but the measured portion of the test will involve 10,000 in operations to remove the tuples. This will be performed using a template
linda.connect()

tup = (1, ) * size
for i in range(10000):
    linda.uts._out(tup)

template = (int, ) * size
start = time.time()
for i in range(10000):
    linda.uts._in(template)
end = time.time()

Figure 8.7: Code For The In Tests

filled only with types. The test uses the code shown in figure 8.7.

The results are shown in figure 8.8. The difference between the registered and traditional versions is 14%, while the unregistered version is 50% slower than the traditional version. The cost of autoregistering types when the \texttt{in} request is made accounts for the majority of the slow down as the it is the only difference between the registered and unregistered versions. The registered implementation is slower because of the expense of generating the one isomorphism (which is cached and reused) and passing the values through the generated isomorphism.

\section*{8.4.1 Complex Types}

The \texttt{out} test was repeated using single element tuples, where that element contains a value of a product type, and the test for the retrieval of data will be repeated in the same manner. The key difference now is that rather than having to compare each element of the tuple the \textsc{Linda} server need only compare one type. The implementations cache the result of the comparison of the types and will, after the first comparison, only need to perform one quick look up per-tuple.

Figure 8.9 shows the results of the test, with the traditional system from the results in figure 8.8 given for comparison. The registered system is tested twice because the system detects when the same type from the same process is used and skips the isomorphism step. The line ‘Registered’ shows the performance when the same type is used to place the tuple into the tuplespace as is used to retrieve it.

The next set of tests uses the same process to perform the \texttt{out} as it does in the \texttt{in}. For the registered version this results in no isomorphism being generated and no conversion of the value being done as the kernel knows that the requesting process is using the same type.

\section*{8.5 Memory Change}

By introducing an isomorphism as the method of determining whether two types match the server will use more memory proportional to the number of different
Figure 8.8: Time Taken To Retrieve 10,000 Tuples

Figure 8.9: Time Taken To Retrieve 10,000 Tuples
types that are matched. In order to measure how much extra memory is used 1,000 tuples will be placed into a tuplespace and the memory used by the kernel measured. A second process will read the 1,000 tuples and then memory used by the kernel will be measured a second time. As the tuples are read and not removed from the tuplespace the difference between the two measurements is the memory used to hold the generated isomorphism. The isomorphism is not garbage collected as the value using the types remains in the tuplespace.

The most obvious conclusion that can be drawn from this graph (figure 8.10) is the fact that for all tests the memory used when matching values is not dependent on the number of values in the product. This is counter-intuitive as the more values that must be converted the larger the isomorphism must be. The memory measured is the memory given to the process by the operating system, which is in a minimum of 4KB blocks, not all of which may be in use. In addition Python’s garbage collector may keep hold of unused memory to avoid the overhead of allocating it from the operating system. This implies that the difference in size of the isomorphisms between the 3 element and 48 element product types is less than 4KB. This makes sense as all the elements of the products are the same, so the overall isomorphism is simply a number of pointers to the same simple isomorphism for converting between the elements of the products.

The traditional version, which does not generate any isomorphisms uses a small amount of extra memory when matching values. This is due to the Python garbage collector not automatically returning memory to the operating system, and the temporary storage of the value being matched forces Python to allocate some extra memory. As is expected the registered and unregistered systems, when not us-
linda.connect()

tup = (0, ) * size

for i in range(10000):
    linda.uts._out(tup)

template = (int, ) * size
ts = linda.TupleSpace()

start = time.time()
linda.uts.collect(ts, template)
end = time.time()

Figure 8.11: Code For The Match Tests

...ing compound types use essentially the same amount of memory as the traditional version. Since the only isomorphism generated in these systems is the identity isomorphism the memory used is minimal.

8.6 Matching Tuples

The previous tests measured both the time taken to match tuples, as well as the time taken to retrieve them from the server. The very first test showed that placing tuples into the tuplespace is marginally slower for the typed systems than for the traditional LINDA. To measure the first of the two operations implied by the in test requires that tuples are not transferred outside of the kernel, but are still matched. Fortunately the implementation of LINDA used supports the collect primitive [26] which takes a template and moves all matching tuples from one tuplespace partition into another.

This test will measure the time taken by the server to match 10,000 tuples against a template, and move them into a new tuplespace. Although the movement into a tuplespace will take some time, it is performed entirely in the server and will be very fast as no copying of the data is required.

The test will be run by outputting 10,000 copies of a tuple containing only integers, then timing how long it takes for a ‘collect’ operation to return, where the template is a tuple full of integer types. The code used is given in figure 8.11.

The results for the different parts of this test are much closer than those in the previous in test. The results have changed little for the registered and traditional versions, around half a second quicker in this test. This shows that the network and the process of serialising and deserialising the messages has a minor impact on the performance of these systems. The unregistered system is much quicker, and is essentially equal in speed with the registered system. This is because only one message is sent, and therefore the expensive type registering operation only needs to be performed once. In this case one application of this operation is far outweighed...
Figure 8.12: Time Taken To Move 10,000 Tuples Between Tuplespaces

by the matching process itself.

The test reinforces what was found in the in test: that while slower in some cases the compression of data into a single value means the LINDA kernel has to do fewer matches which in turn can lead to a speed increase while matching. This test has also removed the expensive registering of types for each message which has shown that the matching performance of the registered and unregistered systems is identical. This is the expected result as the two systems share their type matching code.

8.7 First Retrieval

The proposed system generates isomorphisms when two types are compared for the first time. Once they have been compared the isomorphism is stored and used later. When measuring the time taken to retrieve a large number of tuples this results in the high cost of generating the isomorphism being spread across all the retrievals. In this test only one tuple will be retrieved so the entire cost of generating the isomorphism is reflected in that single operation.

The result in figure 8.13 shows that, as expected the traditional system performs the best for any tuple or product size of all the systems. What is interesting is that the tests using compound types show only a small increase in the time taken to retrieve a tuple compared to the non-compound type versions. This shows that the generation of type isomorphisms is quite an efficient process. As expected the unregistered version is significantly slower than the registered version as it must perform the registration of the types while the registered version does not.
8.8 Solving Sudoku

In section 7.2 a Sudoku solver was introduced. This case study will be used to test the rate at which the different LINDA implementations scale. Scalability is a difficult concept to define, and even harder to measure. The typical runtime performance of the algorithm used in this solver is that a worker process will block until a worker process on a different node generates more than one new grid (as the process that generated the grid will be given the first one). Once it has received a grid the kernel it is connected to will be able to work without communicating with other kernels in the network, until the worker process has exhausted the search space under the grid it obtained originally. If a worker process is starved of work then the kernel it is connected to will communicate with other kernels in an attempt to obtain more work.

The puzzle used in this test is given below. It was taken from The Times on Wednesday February 23rd, and was rated as medium difficulty.

```
  _ _ 9 _ 2 3 1 _ 7
  _ _ 5 _ 6 7 8 _ _
  _ _ _ 8 _ _ _ _ _
  _ 1 _ 4 _ _ 9 _ 3
  _ _ _ _ 5 _ _ _ _
  7 _ 2 _ _ 9 _ 1 _
  _ _ _ _ 5 _ _ _ _
  _ _ 7 9 4 _ 2 _ _
  _ _ _ _ 5 _ _ _ _
```

Figure 8.13: Time Taken To Retrieve One Tuple
Figure 8.14: Time Taken To Solve A Sudoku Grid

The results shown in figure 8.14 show that all implementations follow a similar curve. This suggests that they all scale at a similar rate. The differences in the time taken by any of the five tests is small, and this is more likely because the proportion of time that kernels spend waiting for work is quite high. The unregistered system is shown to be the slowest system, consistent with the previous results. The registered system with compound types is quicker than the traditional system by ten to fifteen seconds across all the network sizes. This is an excellent result that further enhances the findings that larger values in smaller tuples can be quicker than small values in large tuples.

8.9 Many Types

This test will measure the effect that registering increasingly large types with a LINDA kernel has on the time to register the types. As this test focuses purely on the registration of complex types the traditional and unregistered versions of LINDA cannot be used as part of the benchmark.

There are three types that can be used to test the effect that a change of size of a type has on the system, these are product, sum and function types. Product and sum types can be varied by simply increasing the number of component types they contain. Function types on the other hand always have two components, the parameter and the return type. The return type can be a function itself which can be used to extend the size of the function type.

This test measures a number of different elements of both the client and the LINDA kernel. It tests the parser that converts the type information into a syntax
linda.connect()

start = time.time()
for i in range(10000):
    linda.Type("t :: int * %i;" % (size, ))
end = time.time()

Figure 8.15: Code For The Many Types Tests

Figure 8.16: Time Taken To Register 1,000 types
tree. It tests the serialisation of the syntax tree as well as the time taken to store
the type in the kernel and generate a unique id. The graph (figure 8.16) shows that
for both the product and sum types as the size of types increases the time taken to
register it increases linearly. This is exactly the result that is expected and shows
the rare occurrence of registering a new type will not have a significant impact on
the system.

The line for function types shows that registering a function type takes consider-
ably more time to register than a product or sum type. This is caused by the internal
representation of a function containing only an argument and a return type. This
means that for the 48 element function type shown there are 47 functions nested.
With the product and sum types they are nested to only 1 level. Functions with
large numbers of parameters are relatively uncommon, so this should not cause a
significant problem for the programmer.

8.10 Many Systems

Until now all the tests have been run on Amazon’s EC2 network. While this is
a useful platform for running tests because it is consistent and repeatable, it is a
homogeneous network. This thesis was aimed at not only solving problems caused
by different languages but also those caused by different hardware architectures. To
test the system against different architectures it is necessary to move from Amazon’s
EC2 network to a custom network. Resources were not available to test the system
on a wide range of machines. However, the difference in the performance of the
system between the architectures tested here and others will be minimal.

The test uses three different combinations of hardware and operating systems.
One machine is an AMD64 3200+ running 64-bit Gentoo Linux, the second is an
AMD64\(^5\) 4000+ running in x86 compatibility mode running Microsoft Windows XP
SP2 while the final machine is one from Amazon’s EC2 network. The two AMD64
machines are connected through a 54MBit/s Wireless network. This network is
in turn connected to the Internet through an 8MBit/s download/1MBit/s upload
ADSL connection. Communications between the two AMD64s will only traverse
the wireless network while communications between them and the third machine in
Amazon’s EC2 network will use the Internet.

The time taken in four of the five tests (shown in figure 8.17) is very similar.
The network latency caused by communicating with a machine over a public network
(and indeed having to cross the Atlantic to communicate) places a very high delay on
the system and results in the changes in speed due to the differing implementations
being cancelled out. The unregistered system when using compound types shows a
significant decrease in performance, most likely due to the larger number of packets
caused by the large type information that is sent with each message.

---

\(^5\) AMD’s AMD64 architecture is a 64-bit extension to the x86 instruction set. It is possible to
run both 64-bit operating systems on the chip, or for the chip to run in 32-bit mode where it acts
as a 32-bit x86 processor.
While this test does not verify that the system performs the same when run on different hardware architectures it does show the important point that network speed and network congestion can have a huge impact on the performance of a distributed system. In this example the time taken to transfer messages between the different machine dwarfs any processing time. The processor usage was relatively low throughout all the runs, although it was higher on the unregistered system when using compound types, which confirms this.

It has already been shown how the system performs when it has to convert values between different type representations. In these examples the system must generate an isomorphism, then it must split apart a value and put it together in the new format. This is exactly what happens when two systems running on different architectures try to communicate using otherwise identical type representations. The code executed is exactly the same whether the type representations are different because the programmer specified them differently, or they are run on hardware with a different endianness, or different word size.

8.11 Conclusion

In this chapter the two extensions have been tested and compared against each other, and against the traditional version of LINDA. It has been shown that the registered version of the extension performs well against the traditional version in all areas in which they were compared. Although the additional data that is sent increases the network traffic this does not have a significant effect on the performance of the system. The extra processing required for calculating isomorphisms does cause a
slow down, but only when the number of data items that are used in each type is low. When the size of a tuple is reduced at the expense of an increase in the size of the values, and large number of requests are made for that tuple, then the registered version beats a traditional implementation.

The performance of the unregistered system is so poor that it cannot be considered as an alternative to a registered system. Despite the advantage of always having the complete type information accessible from the values, the performance loss is too great. It has been shown that all three systems share a similar scalability and performance in a slow network environment through the tests with the Sudoku solver and multiple hardware architectures.

This chapter has shown that the final requirement of the work in this thesis has been met: that the performance of the system remains practical.
Chapter 9

Evaluation

In this chapter the theory and the corresponding implementation that has been developed will be compared to the goals that were set out at the start of this thesis to determine if they have been achieved. The comparison will look at the various levels that the work dealt with, from the low level representation problems to the changes to the LINDA system and the client programs.

9.1 Representation Of Compound Types

The key motivation behind this work was to make different languages running on a heterogeneous network communicate in a type safe manner as part of an open coordination system, such as LINDA. Initially it was shown how types can be considered to have a number of equivalent (up to isomorphism) representations depending on a number of factors including the compiler used and host architecture and the representation used by the programmer (chapter 3). Types that are implemented in different host languages may have equivalent (again, up to isomorphism) representations but due to the differences in the way languages are compiled and implemented they cannot be identical.

In chapter 2 the work was placed into the context of both the traditional type system of LINDA (single and multi-language versions) and of other distributed systems such as CORBA and DCOM. Systems like CORBA are a very different breed of system from LINDA, they have a steep learning curve and while they have many advanced, enterprise level features they are correspondingly difficult to use. They also use the notion of objects as the prime method of distributing computation, i.e. one process can call a method on an object somewhere on the network in order to trigger computation on a remote (or possibly the local) machine. A coordination system provides a message passing model combined with some storage to allow the messages to be stored rather than forcing them to be retrieved at the moment they are sent.

The methods used for the representation of types in CORBA and DCOM were shown to be impractical for open coordination systems. The architecture of these systems means that the type signatures of objects used are known at compile time.
and thus the compiler knows how the compound type is represented. Although CORBA makes a small gesture towards allowing processes to communicate without knowing in advance what the exact interface is (in that a program can dynamically query an object and build a request at runtime), this is more useful for a runtime debugging tool as it cannot easily be automated.

Representation was introduced in its most natural form to computer programmers, that of the binary ‘in memory’ representation of values. This interpretation of the representation of types has a number of limitations, the principal one being that details such as memory alignment, endianness and even whether the entire type is stored in a contiguous block of memory obscure the main issues when dealing with type representation in a distributed setting. To avoid these details the system requires the programmer to specify types in an abstract form where specific language constructs are converted into combinations of product, sum, function and atomic types.

The conversion process between program code and this abstract representation of the types was introduced, in chapter 3, for two reasons. Firstly, there may be more than one choice when deriving the representation (such as with a mapping). Secondly, because it prevents the implementation from needing to use a preprocessing step on the source-code and this avoids the need to parse the source-code for every language that is being supported. One of the key motivations behind the work was that being forced into using a single language for all client programs, or for the programmer to have to manually decompose types into atomic types, breaks the illusion that the Linda primitives were always part of the language. In this respect the work has not completely achieved the goals that were set out as, in order to use a compound data type, the programmer must perform some additional work over and above placing a value of the type into a tuple. The work they must perform is straightforward and, unlike the process of decomposing values of a compound type into atomic types, the constituent values are kept together as a single group. If the programmer is forced into decomposing values into values of atomic types, and more than one compound value is being decomposed, then the line between the resulting set of values, which divides the two compound values, in the tuple is not well defined. The work in this thesis removes this problem.

The fact that the implementation requires the programmer to create conversion functions to and from their types’ abstract representations is unfortunate. However, is it not a requirement of the work. It would be possible to implement a parser for various languages and generate type coercions from the program code, as was done for the Mockingbird project [124,125]. The additional time that such an implementation would take was prohibitive and thus the concession was made to require the programmer to perform the work manually. Adding such a preprocessing step is likely to require additional annotations to be added to the source code (as is the case in Mockingbird), which are potentially confusing as the system would no longer work with a pure implementation of the host languages.
When used in the same way as any prior implementation of a multi-language LINDA system, the implementation is no worse as atomic types can be used with no extra code, and in languages that do not already provide them, objects representing the atomic types are included to allow matching against them. It is important that the work does not make things worse than they already are, and in this case there is no change.

9.2 Representable Types

In order for the LINDA primitives to appear as part of the language it should be possible to use any and all types expressible in a programming language in a tuple. The larger the range of types that are supported the better and easier it will be for the programmer to use. It is clear that the most obvious types - structs, classes, unions and to some extent functions - can be easily represented in the system. Other, more specialist types (e.g. hash maps or trees) that exist in a language may be trickier to represent, though most can be represented with judicious use of functions.

Functions are perhaps the most useful type, but function values are the most difficult to represent. In chapter 5 the many and varied methods of representing functional values were discussed and compared. The end result of this discussion was that to enable an implementation to be completed in a way that maintained the properties of LINDA (i.e. that when multiple processes read a tuple the values are copied and that a process or host may leave the network and the rest will continue unaffected) functions would have to be implemented using a single language that runs on a virtual machine.

The requirement for the functions to be implemented in this fashion is a concession of the implementation, and methods that would allow functions to be represented in other ways were discussed in section 5.2. However, they were dismissed because of the difficulty of implementation and issues such as global variables and access to the file system. Were the system to use dynamic binary translation or an emulator that could emulate each type of machine on the network on every other type of machine, then functions could be written in any language and compiled into any machine code and still run. This would not solve the problems of allowing a non-pure language to be used and although it would be possible to keep global variables in sync between processes on different hosts the problem of where input/output occurs, especially when the original host has left the network, is not possible to solve.

Despite these problems it is clear from the case studies such as sections 7.4 and 7.5 that function types provide a useful addition to LINDA and enable a range of interesting possibilities for the programmer. Function types are essential for increasing the range of types that are representable in the system. Types, such as a mapping, that have an ‘input’ and an ‘output’ cannot be accurately represented without using a function. Although, using the set notation of a mapping, the type can be represented as a list of pairs this can be interpreted, literally as a list of pairs. By using a function the mapping cannot be misinterpreted.
The work began by studying the product type as it is the most fundamental of all compound types. Unlike functions there are not many issues to deal with regarding the representation of products and thus they can represent all possible implementations of a product type in a real language. C’s structs, C++’s classes, Python and Haskell’s tuples are all product types that can be simply represented using the product type available in the implementation.

The next type dealt with was the ‘sum’ type. A value from a sum type contains a single value from a collection of different types. The problem of representing a value of a sum type is mostly solved by the other types, the only problem was that, as well as the value, a sum type must also store which of the constituent types this value is from. The type of the value is represented simply as an integer which indexes into the list of types that make up the sum type. As the type specification is available wherever a value of that type is, this is an acceptable way to store the information.

Representation of types was a critical foundation of this work. By dividing types into only a few compound types the amount of work required was reduced to a manageable level. With the reduction in the amount of basic types comes a corresponding increase in difficulty for the programmer as they must decompose some of their programming language’s more complex types (such as a mapping) into the types supported by the system. This is not entirely desirable but it is much better than what was previously available as they can keep the structure of their types intact. Again this is a situation where the changes to the LINDA system make it no worse than it was previously (the programmer can still manually decompose complex types to atomic value if they wish) and many cases will be much better as the complex structure of the types used in the programs can be represented.

Subtyping was not dealt with in this work as its relationship with type isomorphisms is not straightforward. A value of one type cannot be transformed into a value of a subtype without creating or throwing away data, depending on the direction of the transformation with regard to the subtype relationship. Subtyping is important when it comes to correctly solving the problem of different sized integers and floating point numbers. The implementation as it stands only matches atomic types when they are of the same size, or in the case of characters, their encoding. Fortunately there is little variation in these, and most hardware uses the same sizes and formats. There is a much wider variety in programming languages, and that is a problem that the implementation does solve. The techniques proposed in this thesis are required to solve the subtyping problem, and the addition of subtyping requires only a modification to the type isomorphism algorithm used.

9.3 Changes To LINDA

Open coordination systems, such as LINDA, were the focus of this work. The introduction of a new type matching algorithm was intended to allow the programmer to use complex types that are defined in their programs without having to decompose them and lose all the type information. This should be done with as little extra
work on the programmer’s part as possible, and still allow the LINDA system to be scalable and efficient.

Two possible methods for implementing the changes to LINDA were discussed in section 6.2. Either a process must register a type before it can be used, or the type information must be sent in full with every communication. Both methods have their own advantages and disadvantages but neither is significantly better than the other based on the analysis before implementation. As a result of this both methods were implemented and used in the runtime analysis that was performed.

From very early on in the analysis (section 8.3) it was apparent that the unregistered version performed substantially worse than the registered implementation. The cost of automatically registering types is extremely high, and performs worse as the number of types that are known by the kernel increases. The cost of generating an isomorphism is even higher, so not automatically registering the types makes the situation even worse.

The registered version is slower than the traditional version (section 8.4), but by treating what would have been several individual tuple elements as a compound value its speed is greatly improved. Although it is hard to describe an average use case for a LINDA-like system it is clear that the larger the amounts of data that are being transferred, and the fewer compound values that can be used to store them, the less of an impact type isomorphisms have on the system.

9.3.1 Client Side

The client programmer, as has already been discussed, must write some conversion functions that convert values from their concrete implementation to a suitable abstract format (section 3.4.1). Other than this conversion function the programmer need not do anything special to use their type. In each supported language the code for registering types or sending the specification with the tuple is hidden and the act of creating a type object which contains the type’s specification and conversion functions creates an object that can be used to encode or decode the type correctly.

The code that is hidden from the programmer is fairly simple for both the registering and non-registering cases. In the case of the registering implementation when a type object is created it transmits the type specification and receives a unique id in return. This id is then sent with every tuple that contains a value of that type. In the non-registering implementation the creation of a type object performs no communication with the server. The communication of type information does not happen until a value of that type is actually transmitted.

When balanced against the goals of this work the changes that are required to the client programs are minimal and therefore the key benefits of LINDA to the programmer are maintained. The two different implementation strategies that were outlined make no difference to the client programmer so the choice of which to use is purely based on their performance.
9.3.2 Server Side

The client processes interface with LINDA through a small, well defined, set of primitives. Although these define the expected interactions with the system there is a large amount of possible variation based on the implementation of the server. Performance, scalability and fault tolerance are all a direct result of the LINDA kernel’s implementation. Any changes that were made to the system as part of this work must maintain, or not seriously degrade, them in comparison to an unmodified system.

The changes that were made to the LINDA kernel can be divided into two groups, the new type representation isomorphisms and the compound type serialisation. The type representation isomorphisms are essentially a drop-in replacement for whatever type equality mechanism was included previously. Although the type representation isomorphism algorithm will undoubtedly be more complex than the previous method for determining equality this will not effect the scalability of the system as it just requires the values and type specification which were already being stored. The isomorphisms that are generated can be thrown away at any time and regenerated as and when they are needed thus the system is not going to 'leak' memory as it runs, nor is it going to utilise more resources over time.

As was demonstrated in chapter 8 the algorithm for generating the isomorphisms is slow. In the case studies it was also shown (section 7.6) that the majority of comparisons are repeated many times thus the addition of a simple cache greatly reduced the impact of the more complex type matching algorithm. Using a cache does mean that the isomorphisms are stored for longer and so the memory usage of the kernel will go up, but only to a certain maximum depending on the limit on the size of the cache.

The scalability of a system is a measure of how well the system can cope with an increase in requests, client programs and kernel nodes. There are a variety of methods for implementing a LINDA system each of which results in different handling of scalability and fault tolerance. This work is not concerned with the details of implementing LINDA from the ground up, its primary focus is on adding correct type matching to an existing system — as is the case with the PyLinda system used for the implementation. In order to be successful the changes in type matching should make the scalability and fault tolerance of the system no worse than it was with a simple type equality matching scheme. As described above although the isomorphism requires more work to calculate than a simple equality the memory required is still constant for a fixed size of type specification.

9.4 Representation Isomorphisms

At the very core of the work is the idea that previous implementations of LINDA and LINDA-like systems have ignored the important question of what makes types equivalent. The concept of an isomorphism is a useful way not only to define a
set of equivalence classes but also to produce functions that can convert between elements of these classes. In the world of types there are many possible different but equivalent representations for every type, depending on factors such as the programming language used, architecture that the code was compiled to run on and the compiler used.

The size of the equivalence class defined by the isomorphism used must not be too large as it would allow types that should not be matched, also it must not be too small otherwise too few things will match. If too many types are matched then types that are not different representations of the same abstract type will be said to be equivalent. The biggest cause of this is the ‘semantic problem’ that is caused by the algorithm only matching on the type signature and not on any meaning for the types. This is no different from how a traditional LINDA system operates so it is still possible that a Cartesian coordinate will be matched with a polar coordinate (as both are represented with a pair of floating point numbers). However, in a system supporting complex types a Cartesian coordinate will not be matched with a tuple containing two separate floating point values.

This work is not making the claim that all ‘incorrect’ type matches will be eradicated. Without solving the (probably unsolvable) semantic problem from section 4.8 this would be impossible. The original method of type matching was still useful despite the existence of semantically incorrect matches and the changes introduced by this work perform no worse than this. In many cases they will reduce the possibility of types being incorrectly matched as compound types need not be decomposed.

Apart from the limitations on the size of the equivalence class defined by the isomorphisms the other main features that were required were that it was easy to retrieve a function to perform the inferred conversion and that it was fast enough to be practical. Although the fact that only one function can be retrieved at a time (i.e. the two functions that make up the isomorphism are not generated as a pair) perhaps does not quite meet the requirement, in reality the conversion will only be one way at any one time. The isomorphism guarantees that if a function is generated in one direction then the inverse function can be generated as well. Even if we do not currently need the inverse function the fact that we can generate it in one direction means that it is there if we do need it.

### 9.5 New Languages

Up to this point the implementation has concentrated on two languages: C and Python. Although these are two quite different language types these are clearly not an exhaustive representation of all programming languages. As has been stated many times the LINDA primitives are designed to be added to any programming language to provide coordination facilities. The work described in this thesis is designed to allow any programming language to use their own compound data types and to receive equivalent types from any other language. In order to demonstrate how easy it is to add support for LINDA to a new language the steps required to give Scheme
access to LINDA are given below. While this is new work it is designed to show that implementing support in a new language does not require any modifications to any of the work shown previously.

Scheme is a Lisp dialect that is unashamedly minimalist. The document that defines standard Scheme is a mere 50 pages long, less than the index to the standard for Lisp. One disadvantage of this simplicity is that there are many implementations each with their own set of libraries, extensions and quirks. The implementation described here is designed to work with Guile\(^1\), the GNU Scheme implementation. Not only is Guile free in every sense of the word but it is also fully standards compliant.\(^2\) Guile also is designed to be easily extended using C extension modules and also to be included into C programs to provide a built-in scripting language, although this second feature is unimportant for this work.

### 9.5.1 Design Decisions

It is important that the primitive operations appear to be as natural as possible to a programmer working in the client language. As Scheme is a dynamically typed, strict, non-pure, functional language this means that the LINDA primitives must not rely on values being of a specific type (as there is no type checking to enforce this). The fact that Scheme is strict (as opposed to lazy) and non-pure is advantageous as LINDA primitives are by definition input/output operators and therefore are non-pure.

The C library used to provide LINDA primitives for C programs implements all the required low-level functionality. Rather than write a pure Scheme library that re-implements the really low-level details such as using sockets to communicate with the kernel the C library will be wrapped with the appropriate code to turn it into a Guile extension module. The important details, such as the organisation of the LINDA primitives and the handling of Scheme types will still have to be implemented, but they can be implemented in either C or Scheme.

The key primitives for LINDA: `connect`, `in`, `rd`, `inp`, `rdp`, `out` and `tuplespace` can all be implemented as simple Scheme functions. The additional functions required to implement the conversion between the concrete types used and their abstract equivalents are also just a set of functions. As Scheme is untyped it is not possible to automatically call the correct conversion functions to serialise values, as is the case with the Python binding and instead the programmer must ensure that they are called at the right moment.

In a previous section (see section 3.4.2) the difference between a tuple and a list was discussed but in Scheme there is only a list constructed by the `cons` function. Ideally the representation of tuples that are to be communicated using a tuplespace would be a tuple type that is built into Scheme. Unfortunately Scheme does not have a specific tuple type, instead it only has a pair. This gives two options for the

\(^1\)http://www.gnu.org/software/guile
\(^2\)At the time of writing the latest version (1.8) is compliant with the latest Scheme standard (R5RS).
implementation, either the pair should be used for tuples or a separate tuple type should be implemented. Clearly implementing a separate type when a similar one already exists is not the best way to ensure LINDA integrates cleanly with Scheme therefore the language binding will expect lists where traditionally a tuple would be used.

Functions are the most common object type in Scheme, but it may be misleading to refer to them as a type. As Scheme is dynamically typed the return type of a function is not specified, nor are the types of any parameters. This means that beyond the fact that it is a function and that it takes a certain number of arguments (even this may not be the case with variable argument length functions) the type cannot be further specified.

Beyond the aforementioned linked list types, and a selection of atomic types the Scheme standard does not specify many types. Indeed there are no language constructs that the programmer can use to explicitly create new types. This conversion can be performed through the use of functions that convert from concrete to abstract types in the usual way. Unfortunately as Scheme is untyped the programmer will have to perform their own runtime type checking to ensure that their functions are operating on values of the correct types.

9.5.2 Writing The Binding

A language binding is divided into two parts. The code that serialises values and communicates with the LINDA server over sockets and the section that interfaces with the language that the LINDA client is being bound to. While there is an argument in favour of writing the binding purely in the language being bound, this is a waste of the programmers time as most languages, including Guile, have facilities to use C extensions to provide extra functions. The advantage of writing the entire extension in the language being bound is that it is ‘cleaner’ as it requires nothing more than the programmer would already be using to run their client program. The vast majority of computers will either have access to a C compiler or can be adequately served using a pre-compiled binary so there is not much additional difficulty in using an extension written in C. By using the C library (which is already written) the amount of code that needs to be written, and as a result the number of bugs that need to be fixed, is greatly reduced.

By writing the extension module in C the majority of the code has already been written. Each of the LINDA primitives becomes essentially a wrapper around a call to the equivalent function in the C library. This leaves only the type handling code and the conversion from Scheme types to values suitable for serialisation to be written. Guile allows a C extension module to store a pointer to an arbitrary C value and have it treated as a native Guile value. This means the type handling functions themselves just become wrappers around the appropriate C calls with the correct conversions for Scheme atomic types.

All previous language bindings that have been described provide some degree of
(use-modules (pylinda linda))

(define (ping-pong)
  (sequence (out uts (list 'ping))
     (in uts (list 'pong))
  )
)

(ping-pong)

Figure 9.1: Scheme ‘Ping’ program.

(use-modules (pylinda linda))

(define (pong-ping)
  (sequence (in uts (list 'ping))
    (out uts (list 'pong))
  )
)

(pong-ping)

Figure 9.2: Scheme ‘Pong’ program.

automation for the conversion of types from concrete to abstract representations. The code for each function checks that any tuple or template that is passed in is actually a Scheme list and that each element is either an atomic type or a Linda abstract type. If the element is an atomic type then it is converted into the equivalent abstract type. The system will also automatically convert some other built in types such as lists and mappings.

9.5.3 Using The Binding

Having implemented the new language binding it is hoped that the design decisions that were made mean that the Linda primitives fit cleanly into the Scheme language. This will be evaluated by implementing three programs in Scheme and seeing how easy it is to use.

The first program to be implemented is shown in figures 9.1 and 9.2. The two client programs simply send out a string, either ‘ping’ or ‘pong’, then wait for the other program to respond. This repeats indefinitely.

The two programs are nearly identical and are very straightforward. The automatic serialisation of values of simple atomic types to their abstract representations is shown to work cleanly. The implementation of the two programs is extremely intuitive. As a second, more complicated, example we will use a Scheme function to create an object that represents a person. The code for this example is given in figure 9.3. The first few lines define a Person object that stores a person’s name and
(use-modules (pylinda linda))

(define (person name age)
  (define (operations op)
    (cond ((op eq? 'get-name) name)
          ((op eq? 'get-age) age)
          (else (error '(invalid operation))))
  )
  operations
)

(define (person-to-pair memo p)
  (product (p 'get-name) (p 'get-age)))
)

(define (pair-to-person memo p)
  (person (toString (getFirst p)) (toInt (getSecond p))))
)

(define person-type (make-type "person: string*int;"
  person-to-pair pair-to-person))

; send a person
(out uts (list (person-type (person 'Andrew 24)))))

; retrieve a person
(person-type (in uts (list person-type)))

Figure 9.3: Scheme ‘Person’ program.

age. The code then defines two functions, person-to-pair and pair-to-person that convert between a pair of a string and an integer and a person object. The functions are relatively straightforward and need no further explanation beyond what was given to describe the equivalent functions in other languages.

The final two expressions place a Person object into the uts and then retrieve it again. Because of the untyped nature of Scheme it is impossible to call the conversion functions person-to-pair and pair-to-person automatically, as is the case in the Python implementation. This forces the programmer to call the appropriate function manually, but this is a limitation of Scheme.

As a final experiment with the Scheme binding one of the case studies will be implemented using it. The chosen case study will be the Sudoku solver previously described in section 7.2. While not as complicated as the later case studies it does use an interesting array of types. The Sudoku grid is a nine by nine grid of either numbers or blanks. In the other languages describe special care had to be taken to handle the case where a square was either a number or a blank. As Scheme is untyped this is a non-issue and nothing special needs to be done. The definition of the Scheme type that will be used to describe the Sudoku grid is given below.
(define (sudoku-grid)
  (let ((grid (list
               (list nil nil nil nil nil nil nil nil nil)
               (list nil nil nil nil nil nil nil nil nil)
               (list nil nil nil nil nil nil nil nil nil)
               (list nil nil nil nil nil nil nil nil nil)
               (list nil nil nil nil nil nil nil nil nil)
               (list nil nil nil nil nil nil nil nil nil)
               (list nil nil nil nil nil nil nil nil nil)
               (list nil nil nil nil nil nil nil nil nil)
               (list nil nil nil nil nil nil nil nil nil)))))

(define (get x y)
  (list-ref (list-ref grid y) x)
)

(define (set x y value)
  (let ((row (list-ref grid y)))
    (let ((newrow (append (take row (- x 1))
                           value
                           (drop row x)
                           )))
      (set! grid (append (take grid (- y 1))
                         newrow
                         (drop grid y)
                         )))
  )

(lambda (command)
  (cond ((eq? command 'get) get)
        ((eq? command 'set) set)
        ((eq? command 'grid) grid)
        (else (error "Invalid command"))
  )
)

(define (grid-to-value grid memo)
  (linda-value (grid 'grid))
)

204
(define (set-loop grid value x y)
  (if (> x 8) nil
      (sequence ((grid 'set) x y (list-ref value x))
               (set-loop grid value (+ x 1) y)
      )
  )
)

(define (row-loop grid value y)
  (if (> y 8) nil
      (sequence (set-loop grid (list-ref value y) 0 y)
               (row-loop grid value (+ y 1))
      )
  ))

(define (value-to-grid value memo)
  (let ((grid (sudoku-grid)))
    (sequence (row-loop grid value 0)
              grid)
  )
)

(define sudoku-type (make-type "sudoku: ((int + Nil) * 9) * 9;
                                      grid-to-value value-to-grid)

9.5.4 Evaluating The Binding

As has been shown over the previous two sections creating a binding for Scheme
to the modified LINDA system is a straightforward process. Although much of the
socket manipulation work was avoided by wrapping an already implemented C li-
brary, the question of how to integrate the proposed type system with Scheme was
an interesting one. The lack of a specific ‘tuple’ type was circumvented by using
the ‘list’ type rather than trying to create a new type. Other than that minor
consideration the language binding is a natural one.

Although designing the binding brought up a few interesting questions due to
Scheme’s untyped nature there were obvious and natural answers to them. The
biggest question was how to handle the fact that custom compound types are typ-
ically implemented using functions. There was no option but to require the pro-
grammer to implement the ability to retrieve the current state of the object and to
be able to recreate the object from the state in addition to creating the functions
that perform the conversion between their types and the equivalent abstract rep-
resentation. Most types will already be built with the ability to save and restore
an object’s state so this first requirement is not too big a burden and the second is simply a continuation of the requirements placed on all client programmers.

The automatic conversion of atomic types to their abstract representation is a useful, but not vital feature to have in a language binding. With the Scheme binding this simply involved adding a few extra checks to the code that converts from Scheme lists into tuples which are used for communication. As lists are used to represent tuples it poses an interesting problem as to how they should be handled with regard to the automatic conversion. As has been demonstrated in previous chapters tuples and lists have different representations (i.e. tuples have a fixed size, lists are typically variable length). Fortunately Scheme gives a simple solution to this problem: if the final element of a sequence of pairs is a ‘null’ then it is a list, if it is an atomic object then it is a tuple.

9.6 Using Other Systems

This chapter has dealt exclusively with the thesis’ contribution in the context of an open LINDA system. One of the motivating factors for this work was that many systems use methods for avoiding type representation problems that place unnecessary burdens on the programmer. Despite concentrating on open coordination systems the process of using type representation isomorphisms to determine type equality is applicable in both closed coordination systems and general distributed computing systems.

Closed coordination systems share a great deal with open systems, except that they require all code to be available and compiled at the same time. They are still designed so that many programs work together to solve a problem, only the additional restriction of having to know what other programs will do allows the compiler to perform many extra static optimisations over and above what is possible in an open system.

A general distributed computing system such as CORBA is not usually designed so that many processes collaborate to perform the work. Instead they take the view that some processes are ‘servers’ which export methods that may be called by other, ‘client’, processes to cause the server to perform some work. Despite this difference in approach the systems still revolve around sending typed values between processes. These communications will require the values to be marshaled into a common format decided upon prior to the compilation of the programs. If the type representation isomorphisms were used then this common format would not need to be decided upon and processes would be able to change their representations without having to recompile the entire system.

In this section we shall give a brief overview of the challenges and problems that would be faced were type representation isomorphisms to be integrated into CORBA and a closed LINDA-like system.
9.6.1 CORBA

CORBA forces the programmer to create an IDL specification which is shared between all processes at compile time. The attempt that CORBA makes to avoid the requirement that the IDL file is shared between processes is practically useless if CORBA is to be used as an open coordination system. Although the work was not focusing on this it is perfectly possible to replace the methods for querying an interface at runtime with a method that accepts an arbitrary type and, through type representation isomorphisms, calculates whether they are equivalent or not. This would of course require both sides of the call to provide type specifications and the conversion functions necessary to create a type object.

9.6.2 Closed Linda

Throughout this work we have focused on open coordination systems. Open coordination systems provide an interesting challenge because nothing is known about any of the communications that will happen until they actually occur. With a closed system it is possible to deduce much more about the communications as the compiler has access to all of the source code that will be used.

When compiling the group of programs that make up a closed coordination system the preprocessor still needs to make a decision on which types are equivalent and provide conversions between the different representations of the types. In most cases the closed system will require that the source-code that defines the types is shared between all processes. If the type specifications are used as has been described then it would be possible to run the type representation isomorphism at compile time and to emit the generated isomorphisms as compiled code that can be statically linked with the executable client programs.

This would allow a closed system to use multiple programming languages and multiple architectures as there is an additional abstraction added to the system. The main cause of the slow down experienced in the open LINDA implementation is nullified as the type representation isomorphisms are generated at compile time and not as and when needed. Although there would be some additional expense in running values through the generated isomorphisms this would be fairly insignificant as the code would be statically compiled instead of a dynamically constructed function. Finally, as every possible tuple’s type signature is known at compile time the signature can be reduced to a simple identification number avoiding the overhead of the transmitted type signatures suffered by the non-registering open LINDA implementation that was described previously.

Although this remains as future work it is clear that using type representation isomorphisms statically in closed coordination systems will help provide more flexibility to what is a fundamentally restrictive system.
9.7 Future Work

There are three potential avenues for work based on this thesis: more classes of types, tighter integration with host languages and integration with new systems.

Adding subtyping to the enhanced LINDA system proposed in this thesis would allow the transfer of complete class hierarchies and would solve problems related to different sized integers. The problems related to different sized integers could be approached from two different directions. Either a method could be found to create or destroy information from values whose range is expanded or contracted, or the conversion of values could be halted when it is discovered that there is ambiguity in the result. The transfer of class hierarchies can also be approached in two ways. Either a value of one type can only be converted into a value of another type from the same place in an isomorphic type hierarchy, or a value can be converted to a super or sub-type in the isomorphic hierarchy. The latter of these two possibilities would require the creation of or throwing away of data while the former would not.

Tighter integration with host languages would involve looking at converting type specifications from the host language into a format suitable for communication with the LINDA kernel. It would also involve looking at wrapping types that are built into the host language more cleanly so they can be used much more simply than at present. Types such as Python’s dictionaries are currently awkward to use and are not directly supported by the system.

The previous section described how the type isomorphism algorithm and MINIMAL language could be integrated with new systems such as CORBA or a closed implementation of LINDA. The addition of these elements of the system developed in this thesis to a different system would radically alter the new system and combine the benefits of decoupled client program developments of open LINDA with the benefits of a closed system (such as easier optimisations through knowledge of the system’s performance prior to runtime).

9.8 Conclusion

In this chapter the work has been compared to each of the original criteria that were set out in the introduction. In each case the work was shown to meet the goals that were set out or in the few cases where the exact goals were not met the reasons were shown to be purely due to limitations in the implementation that could be avoided. These were left to ensure that the work being performed by the system is readily apparent and not hidden from the programmer.

The biggest flaws in the system relate to the inability of the implementation to automatically convert the programmer’s type definitions to the abstract representation used in the system. Were the system able to automatically perform this conversion then the programmer would not have to do anything to use their complex types for coordination. The other significant problem is that function types are only representable by a new language called MINIMAL. As was discussed in some
detail, this is because the language must be able to run on any hardware, have functions called from any external programming language and be pure. Function types were used extensively as part of the case studies and were shown to be a valuable addition.
Chapter 10

Conclusion

This thesis began by introducing the concept of an open coordination system. While similar to many other distributed systems an open coordination system imposes some restrictions that differentiate it from CORBA, DCOM and other similar systems. An open coordination system requires that any process must be able connect at any time, and there does not need to be communication between the processes prior to them connecting to the system. The fact that any process can connect means that ideally a process written in any language and running on any operating system and hardware architecture would be able to connect.

As shown in section 2.1 typical implementations of an open coordination system take one of two routes. Either they require that all processes are implemented in languages such as Java, or that processes can only communicate using a small set of predefined types. Neither of these solutions are ideal as one language cannot be the ideal choice for solving all problems, and programmers defining their own set of compound types is a key element of a good programming technique. It was the aim of this thesis to overcome both these problems by allowing programmers to use compound data types across multiple languages, without having to share information between the processes before they connect to the system.

The thesis began back in chapter 1 with a discussion on how values of types are represented in a computer’s memory. This was continued in chapter 3 with a discussion on why issues such as differently sized integers, byte ordering and character encoding schemes mean that the same value can be stored in many different ways by a compiler. Different programming languages have different ways of dealing with types, which introduces an additional layer of complexity to the problem. C treats variables as having a type, with the value as a simple bit string. If the same area of memory is accessed through two variables of different types then the same bit string is interpreted in two different ways. Python is different and associates type information with values, leaving variables untyped.

Most languages feature a wide array of built in types, which differ from language to language. In order to allow types from different languages to be compared a small set of types were focused on. These were products, sums, pointers and functions. Types that are created in a programming language need to be specified using this
small set of types in the type specification language that was introduced in section 3.4. It is possible to automatically convert from the type specifications in a programming language (where one exists) to this language in order avoid the complexity of requiring the programmer to specify their types twice. This is left as future work as hiding the type specification would create confusion when attempting to study the matching of types.

Since a key feature of an open coordination system is that processes do not have to be running on the same physical machine it must be possible to transfer values between machines. The introduction of pointer types to the system means that it becomes possible to also transfer cyclic values. For the same reasons as the type specifications were made explicit the programmer is also required to provide two functions to guide the serialisation of the values.

The type specifications that are defined by the programmer describe a representation of a type. The concept of a type isomorphism was introduced in chapter 4, which is a pair of functions that convert values between two different type representations. The key aspect is that although the type representations may not be identical, they are both representations of the same abstract type. Several different type isomorphism algorithms were presented to illustrate the different approaches taken by type isomorphisms, and to show the different sets of types that can be considered to be equivalent. As none of the three algorithms presented matched the criteria required for them to be used in an open coordination system a modified version of one of the algorithms was selected.

The conversion of values between different representations is, in most cases, straightforward once a type isomorphism has been created. In the case of functions this is not true and in chapter 5 special measures were taken to allow them to be treated the same as any other value. While ideally a function created in any programming language would be able to be transferred to another processes and executed this is not practical. Instead a simple programming language was implemented which integrates well with the type specification language, and allows functions to be called from any language.

Chapter 6 introduced two possible implementation strategies for integrating type isomorphisms with LINDA. Either a type can be ‘registered’ (section 6.4) with the LINDA kernel and a unique id used whenever the type information is needed, or the full type information is sent with every value (an ‘unregistered’ implementation, section 6.3). While sending the full type information leads to a simpler implementation the optimisations that can be performed when each type is given a unique id help keep the system usable and in some cases even out perform the unmodified version.

A series of case studies were introduced in chapter 7 to demonstrate how the improved type system can be used by a programmer. They showed that by condensing values into a single tuple element the work load of the programmer is greatly reduced. The programmer no longer needs to consider the values in a tuplespace and the values they are using in their program as separate, but that they are identical.
values which are converted between automatically.

Chapter 8’s tests on the speed of the system showed that while generating a type isomorphism is an expensive process, caching the isomorphism means that this cost becomes negligible when enough comparisons are performed. The memory usage of the system is increased but not significantly, and the network usage also goes up but this does not affect the performance greatly.

While the type system introduced in this thesis does not integrate as cleanly with the host language as is possible, this is done deliberately and illustrates the work that is necessary to make the system function correctly. Using compound data types has been shown to provide an increase in flexibility for the programmer, and leads to a number of novel solutions that would otherwise be impossible. This is a new and exciting innovation which greatly helps programmers who use LINDA and LINDA-like systems in heterogeneous hardware and language environments.
Appendix A

Using Virtualised Servers To Test Distributed Systems

In order to test a distributed system it is necessary to run it across a large number of nodes. Running the system on just one node will allow certain aspects to be tested, but would leave out a key part of the tests - how the system handles the movement of data between kernels. In many cases getting exclusive access to a large enough collection of computers to test such a system can prove difficult, if not impossible.

A.1 Motivation

Strict demands are placed on hardware by businesses as the more their hardware is utilised the lower their overall costs. If several servers can be consolidated into one physical machine with little or no loss of performance then the cost of additional servers is reduced.

A common situation is for a user to be running one operating system and to require a different operating system to run some software that is not available on the system they are using at that point in time. Or perhaps they are developing a program and wish to test it in a different environment. One solution is to have many computers, one for each required environment. This is both expensive and slow as the user will have to move both themselves and their data between machines. Alternatively they may ‘dual boot’ which is where the user is given the option of which operating system to run when the computer is switched on. This is slow, in that the user has to reboot to access the other system and they cannot switch easily back-and-forth between the two. Finally, they may run one operating system as a host and then run the rest through emulation or virtualisation as a ‘guest system’.

These scenarios, and others, have lead to the development and popularisation of emulation and virtualisation technologies. Virtualisation itself is not a new concept, and was initially demonstrated by IBM in their CP-40 research system and then popularised in their CP/CMS operating system. Each CP/CMS user was given the appearance of a stand-alone machine, even though their session was being multiplexed with other users.
Unless the system is specially designed for virtualisation (as was the case with the CP/CMS) there will be some overhead in the emulation of various system components and until recently this overhead was too great to outweigh the savings of consolidating servers. Processor power has exploded and now processors are easily able to handle several moderately loaded servers at once.

Popek and Goldberg’s 1974 paper [126] introduces a set of sufficient conditions that enable an architecture to efficiently support full system virtualisation. The most common hardware platform today, x86, does not meet the requirements and so was, until recently, not capable of native virtualisation. In recent versions (from Spring 2005 for Intel and from May 2006 for AMD) of their CPUs both Intel and AMD have introduced incompatible but comparable extensions to the x86 instruction set which meet the Popek and Goldberg conditions. The addition of these instructions and the commodity nature of x86-based machines has made virtualisation a practical, efficient and cheap method for running a large cluster of computers.

A.2 Virtualisation Software

A huge range of software is available for virtualisation. There are commercial, free and open source implementations to choose from. Some software, such as QEMU [127], allows you to emulate completely different CPUs to the one it is being run on. This is emulation rather than virtualisation, and imposes a high overhead compared to native virtualisation, but it is perfectly possible to emulate the same CPU as the software is being run upon. QEMU provides a special module that in this case gives it the ability to perform near-native virtualisation. The most well known virtualisation system is that produced by VMware [128]. A commercial offering, VMware’s software gives excellent performance even if it is not using Intel’s or AMD’s extensions to provide true native virtualisation. The final piece of software that will be introduced here is Xen [129]. Xen is an increasingly popular choice and was designed from the ground up to allow commodity operating systems to share conventional hardware in a safe and resource managed fashion [130].

A.3 Testing Requirements

In order to test a distributed system it is necessary to have a standardised and consistent environment. Virtualisation technologies give users the ability to decide exactly what systems they want in their network, how many they want and what specification they should be. It must be possible to reproduce results with little or no variation. This obviously depends on being able to exclude outside influences from the system being tested.

QEMU allows users to run any of six different architectures (with four more in testing) on any host architecture. QEMU runs as a normal user process and so it is not possible to guarantee the system being emulated a slice of the host processor’s time. This limit means that unless QEMU is the only significant user process running
it is unsuitable for use in a testing environment. VMware’s superior performance to QEMU means that it is better, but it is still just a user process running on a machine and so does not have the guaranteed performance required for repeatable testing.

Xen allows one computer to provide many nodes of the same architecture, with well defined and strict limits on CPU usage, disk performance and memory usage. These strict limits allow the virtualised nodes to be run with their performance characteristics defined at boot time and which are consistent throughout the node’s lifetime and are repeatable across runs. The Xen hypervisor is part of the host machine’s kernel and therefore has greater control over the performance of the virtualised machines. The performance of a single virtualised machine is independent of other activity on the node so even if the tester does not have control over what else is being run on the node, Xen still guarantees the performance of the node.

**A.4 Amazon’s EC2**

An ideal situation for testing is that to have access to a large number of computers. Virtualisation can allow an increase the number of nodes available, or to guarantee the performance of the nodes if the tester does not have exclusive access to the machines. It maybe the case, and was the case for the testing of the work presented in this thesis, that the tester do not have access to a sufficient number of instances. The only option is to use a commercial system to provide the necessary computing power.

At the time of writing Amazon are the only company to offer a service which is suitable for this, although with the growing acceptance of virtualisation and expense of setting up and running your own clusters it is likely that more will appear. Their EC2 service allows the creataion of nodes in a matter of minutes, giving the tester full root access to each. The service provides the equivalent power to a 1.7GHz x86 processor with 1.75GB of RAM, 160GB of local disk space, and 250MB/s of network bandwidth.

EC2 is based on Xen so the performance characteristics are both repeatable and reliable, perfect for a testing environment.

**A.5 Issues**

Even if there is guaranteed processor power, memory and disk space for each of the nodes, unless a cluster of machines separated from any network is being used, it is likely that it will not be possible to control the network bandwidth available. Experience has shown that this is not a problem due to the powerful network used in Amazon’s data-centres.
A.6 Conclusion

This section has introduced the concept of virtualised servers. Both the benefits and drawbacks of using such systems for the testing of distributed systems were discussed. The ease of adding nodes to the network, and reduced cost of not having to provide physical hardware for them to run on are big advantages. The difficulty in guaranteeing the performance of such systems, unless it is specifically designed to provide it (such as Xen) or if the virtualised system is the only user process running on the host system, means that careful thought must be given to ensure performance results are reliable.

The ability of a tester to get exclusive access to a large number of machines when otherwise the number would be much less, or where the access would not be exclusive is a big advantage for virtualisation. With systems such as Amazon’s EC2 service, or similarly well designed set-ups, there are no significant disadvantages to using virtualisation technology.
Appendix B

The Minimal Language

The Minimal language is a simple, functional, strict language designed to fulfill a very specific role. As was discussed in chapter 5 a single language is required which can be used in many different languages and whose functions can be packaged and transferred between separate processes. This packaging must be a true package if the code and all related information and not be the creation of an identifier which causes all calls to the function to be routed back to the original process.

The BNF Grammar for the Minimal language is given in figure B.1. The language’s syntax takes after Haskell, although Minimal is strict while Haskell is lazy.

Below a simple function, double is given, which takes an integer and doubles it. The definition consists of two parts the type of the function, and the function code. The type definition is identical to that which is used with the type specification language. The next line uses the same function name as the type so the compiler can associate the type with the definition, and it gives the single parameter the name x. The rest of the definition is self explanatory.

double :: int -> int;
double x = 2 * x;

Minimal uses a reference garbage collector capable of detecting and deleting cycles to manage memory so the programmer does not need to. To create a pointer to a value the programmer simply calls the Ptr function.

cycle_type :: int * ptr(cycle_type);
cycle :: Nil -> cycle_type;
cycle = let x = (1, Nil) in
    let x[1] = Ptr(x) in x;

The standard Linda primitives are available as functions in Minimal which is what allows the eval primitive to be implemented using Minimal. Sample code is shown below.
Figure B.1: BNF Grammar for MINIMAL
linda :: Nil -> Nil;
linda = let tup = in(uts, (int, int)) in
        out(uts, tup)

B.1 Values

MINIMAL has support for a small range of built in values. Integers and floating point numbers are defined using straight numerical values. Strings are any combination of letters, numbers and characters enclosed in double quotes. Tuples are defined by enclosing a comma separated list of other values in brackets. A single element tuple requires that the comma be included, as in (1, ). A zero element tuple is defined as ()

The MINIMAL language includes support for Nil, which indicates the absence of a value, True and False which have their usual boolean meaning. All values can be coerced into booleans and False, Nil, 0, 0.0 the empty string and empty tuple are all treated as false. All other values are true.

B.2 Primitives

Below is the list of the primitives that are available to the programmer in MINIMAL, and a description of how they are evaluated by the interpreter.

- func x = y
  
  This construction creates a new function called func, which accepts one parameter, x. When the function is evaluated the code y is executed. Before this function can be evaluated the type definition of the function must be given to the interpreter. Evaluating a function that does not have a type specification attached results in an error.

- if x then y else z endif
  
  When evaluated the code x is run and its value is tested as a boolean. If it is equivalent to true the code y is run, if not then z is run.

- let x = y in z
  
  let creates a new scope in which the variable x is given the value that is the result of evaluating y. The code z is then evaluated in this scope, and the result of evaluating that code is the result of this expression.

- do x then y
  
  This expression evaluates the two expressions x and y in turn, discarding the result of x. The result of the whole expression is the result of evaluating y.
• $f(x)$

This evaluates the expression $x$ passes it as the parameter to the function $f$. $f$ itself may be an expression, provided it results in a function. Trying to evaluate any value which is not a function results in an error. **Minimal** supports currying of functions so it is not an error to ‘evaluate’ a function with fewer arguments than it actually accepts. In this case a new function value is returned which remembers which parameters have already been fixed. If a function with $y$ parameter is evaluated with $x$ arguments, where $x$ is greater than $y$, the first $x$ arguments are used to evaluate the function. The remaining arguments are used to evaluate any resulting value, and so on. If a function returns a value which is not a function and there are arguments remaining to be used then an error occurs.

• $t[x]$  

This expression is used to index into a tuple. First the expression $t$ is evaluated, then $x$. If the result of evaluating $t$ is not a tuple or $x$ does not result in an integer then an error is thrown, otherwise the result of the expression is the $x$th item of the tuple $t$.

### B.3 Semantics

**Minimal**’s semantics have been left largely as an implementation detail because the aim of this thesis was not to produce a new, fully fledged, language. In this section each of **Minimal**’s primitives will be taken in turn and a brief description of their semantics given.

**let** expressions take the form `let variable = expr1 in expr2`. **expr1** is evaluated first and the result assigned to the variable which is included in the environment when **expr2** is evaluated. The return value of **expr2** is the value of the entire expression.

Similarly **do** expressions take the form `do expr1 then expr2`. This is useful for functions with side effects that return no values, currently just `out`. First **expr1** is evaluated and any return value discarded, then **expr2** is evaluated. The return value of **expr2** is the value of the entire expression.

Conditionals are implemented using the expression `if test then expr1 else expr2 endif`. First **test** is evaluated, coerced to a boolean and if true **expr1** is evaluated, otherwise **expr2** is. Because types are only checked at entry and exit of functions **expr1** and **expr2** do not need to return values of the same type. The function **if_test** works because the return type is a sum, and the two branches of the **if** expression return values of types that match the sum type.

```
if_test :: bool -> int + string;
if_test x = if x then 1 else "abc" endif;
```
MINIMAL supports the standard arithmetic operations, add, subtract, divide and multiply which are implemented for integers and floats. If an operator is used with an integer and a float as an operand then the integer is converted to the equivalent float and the return value is also a float.

Function calls are implemented by looking up the name of the function in the current closure. Next the runtime evaluates each of the arguments in turn. After evaluation the type of the value is compared to the type given in the function declaration for that argument. If they do not match then an error is raised. Once all parameters have been evaluated a new closure is created with variables representing each argument included. The body of the function is executed in this closure. The result of evaluating the function is the result of evaluating the body of the function in the closure.

B.4 Memory Management

The implementation of MINIMAL as used in this thesis has a cyclic reference counting garbage collector to ensure that no memory is leaked during the running of the program, and that the programmer does not need to concern himself with memory management. Internally each type contains a method which returns a pointer to each of the values which a value of that type references. When a value is deleted (usually by a variable pointing to it going out of scope) it decrements the reference count of each of the other values it contains. In a tuple this means decrementing the reference count of each of the elements in the tuple, a pointer decrements the count of the value it points to etc.

When a reference count is decremented so that it reaches zero the value is deleted, and it decrements the reference count of all values it points to. If the value has not reached zero then the system attempts to find out whether a clique of values has been created which is not externally accessible, so that all the values in the clique can be deleted. To do this the system explores the tree of values referenced by the initial value keeping track of how many times a value is referenced by other values in the tree. A clique is then found starting with the initial value and then expanded by adding all values who reference a value in the clique. If any value of the clique has a reference count greater than that which was calculated then it is externally accessible. Once the clique has expanded to enclose all values that point to any value inside the clique, and all references to values in the clique are from values also in the clique, then the clique is garbage and all values are deleted.

This garbage collection all happens automatically and behind the scenes, so no action is required on the part of the programmer.

B.5 Type Checking

The current implementation of MINIMAL is dynamically type checked which means that type errors are not picked up until runtime. As MINIMAL functions are declared
in a sequence of calls to the Minimal API there is no single time when type checking would be appropriate. Types are checked on the entry to, and exit from, functions. A function such as the one below will run as far as returning the value when it will be discovered that the value does not match the type required of the return value, and the Minimal runtime will signal an error to the host language.

\[
\text{crash} :: \text{Nil} \rightarrow (\text{int}, \text{string});
\]
\[
\text{crash} = (1, 2);
\]

Assigning types in this way leads to a simple implementation but one that has some quirks that the programmer may need to be aware of. Sum types are stored as a value and an index into the sum which determines which of the possible types the value is of. When assigning a value to a sum type the implementation compares the value to each possible type in order until a match is found. If no match is found then a type error occurs.

\[
\text{sumtype} :: \text{Nil} \rightarrow \text{int} + \text{string};
\]
\[
\text{sumtype} = "abc";
\]

\[
\text{sumtype\_crash} :: \text{Nil} \rightarrow \text{int} + \text{float};
\]
\[
\text{sumtype\_crash} = "abc";
\]

**B.6 Conclusion**

This chapter only gives a very brief overview of Minimal. The complete source code can be downloaded which contains additional examples of Minimal code and integration with other languages. See appendix C for instructions.
Appendix C

Getting And Running The PyLinda Code

The PyLinda system described in this thesis is available as an open source project released under the GNU GPL [131] license. It is divided up into three components which together form a complete system capable of running as a traditional implementation of LINDA, and both the registered and unregistered extensions described in chapter 6.

C.1 Getting And Building PyLinda

The latest version of the software can be downloaded using the Subversion [132] source control tool. Simply type the following command into an open shell window and the complete source code will be downloaded to your machine.

```
svn checkout http://pylinda.googlecode.com/svn/trunk/ pylinda
```

The source code can be found in the `pylinda` directory, and is split into three subdirectories - one for each component of the system. Before PyLinda can be compiled some prerequisites must be installed. PyLinda requires `flex`, `bison`, `libxml2`, `Python` and `scons` in order to build and run.

The first component to be built is an implementation of the `MINIMAL` language. Building it is as simple as running the command `scons` from within the `minimal` directory. Once the software has finished building, the library files `libminimal++.so` and `libminimal.so` need to be moved or linked to a location which will be automatically searched by compilers. On a Linux system this will usually be `/usr/local/lib`. The header files `src/minimal.h`, `src/minimal.c.h` and `src/minimal.cpp.h` should be moved to `/usr/local/include`.

Next the library which provides the LINDA interface to client programs must be built. Simply typing `scons` when in the `liblinda` directory will build a library which uses registered types. To build a different version of the library the parameters

---

1Building the software on Windows will generate files with similar, but different names.
types=1|0 and register=1|0 can be passed to scons. The kernel will select the implementation of LINDA it uses based on the parameters used to compile the client library. As with the MINIMAL library the various shared libraries that are generated must be copied or moved to a public directory. The header files src/linda.h, src/linda_c.h, src/linda_cpp.h and src/linda_config.h must also be copied to a public location.

Finally, the source code for the LINDA kernel must be installed. As this is written in Python it does not need compiling, and running python setup.py install is enough to get the system installed and ready to run.

C.2 Running PyLinda

Running PyLinda consists of two steps, firstly the kernel or kernels are run and secondly the client processes connect to a kernel. A PyLinda kernel is started by running linda_server on each machine on which a client program will be run. To connect the kernels together, each kernel after the first should be run with -c <ip address> as a parameter. The IP address given should be that of a kernel which is already running and connected to the system. More command line options are available and can be listed by using the --help option.

A client process automatically connects to the kernel running on the same machine as it, so no configuration is required. To run a Python based client no compilation is required, but a C client must be compiled and linked to the correct libraries. To correctly build a C client the following (GCC) command line may be used -llinda -lminimal -lpthreads -lreadline.

C.3 Problems or Bugs

If problems are encountered while building or running the software the PyLinda mailing list (http://groups.google.com/group/pylinda) or bug tracking software (http://code.google.com/p/pylinda/issues/list) may be used to report the problem and to find a solution.
Appendix D

Case Study Code

In this appendix the complete source code for the case studies from chapter 7 is presented.

D.1 Prime Sieve

This section contains the complete code for the case study presented in section 7.1.

D.1.1 Master

This section contains the code for the controlling process in the prime sieve case study.

#!/usr/bin/python

import sys
import linda

linda.connect()

prime_ts = linda.TupleSpace()
linda.uts._out(("prime_ts", prime_ts))

work_ts = linda.TupleSpace()
for i in range(int(sys.argv[1])):
    linda.uts._out(("work_ts", work_ts))

work_ts._out((2, 0, 0))

for i in range(3, 1000):
    work_ts._out((i, 2, i-3))

while True:
    tup = prime_ts._in((int, ))
    if tup is None:
        break
    print tup[0],
D.1.2 Slave
This section contains the code for the slave process in the prime sieve case study. This process controls a collection of prime sieve processes.

```
#!/usr/bin/python

import os
import sys
import linda

linda.connect()

_, work_ts = linda.uts._rd(('work_ts', linda.TupleSpace))

while True:
    # Get a new prime
    tup = work_ts._inp((int, 0, int))
    if tup is None:
        break
    # Spawn a sieve process
    if os.fork() == 0:
        os.execlp('python', 'python', 'prime_sieve.py', str(tup[0]))
```

D.1.3 Prime Sieve
This section contains the code for the sieve process in the prime sieve case study.

```
import sys
import linda

linda.connect()

prime = int(sys.argv[1])

# Let the master know we've found a prime
_, prime_ts = linda.uts._rd(('prime_ts', linda.TupleSpace))
prime_ts._out((prime, ))

_, work_ts = linda.uts._rd(('work_ts', linda.TupleSpace))

next_prime = 0

count = 0
outcount = 0
while True:
    tup = work_ts._inp((int, prime, count))
    count += 1
    if tup is None:
        break
    # check if the number being tested is divisible by us
    if tup[0] % prime == 0:
        continue
```

228
elif next_prime == 0:
    # this is the first possible prime we’ve found
    work_ts._out((tup[0], 0, 0))
    linda.uts._out(("work_ts", work_ts))
    next_prime = tup[0]
else:
    work_ts._out((tup[0], next_prime, outcount))
    outcount += 1
print "done"

D.2 Sudoku Solver

This section contains the complete code for the case study presented in section 7.2.

D.2.1 Traditional Version

The code in the following two subsections forms the complete Sudoku solver case study which is capable of being run on

Master Process

#!/usr/bin/python

import sys
import time
import linda

def master():
    grid = loadFile(sys.argv[1])

    print "Solving..."
    print gridString(grid)

    empty = sum([len([x for x in row if x is None])
                 for row in grid])

    if not linda.connect():
        print "Please start the Linda server first."
        return

    start = time.time()
    linda.uts._out((empty, ) + gridToTup(grid))

    tup = linda.uts._in((0, ) + (int, ) * (9*9))

    print "Solved Grid in %f seconds." % (time.time() - start)
    print gridString(tupToGrid(tup[1:]))

def loadFile(filename):
    fp = open(filename, "r")
    rows = []
for line in fp:
    if len(rows) == 9:
        break
    cols = line.strip().split(' ')
    if len(cols) != 9:
        raise SystemError, "Row (%s) only has %i columns."
        % (line, len(cols))
    cols = [(x != "_" and int(x)) or (x == "_" and None)
    for x in cols]
    rows.append(tuple(cols))
    if len(rows) != 9:
        raise SystemError, "File only has %i rows." % (len(rows), )
    return tuple(rows)

def tupToGrid(tup):
    assert len(tup) == 81
    grid = []
    for r in range(9):
        grid.append([x != -1 and x or (x == -1 and None)
        for x in tup[r*9:(r+1)*9]])
    return grid

def gridToTup(grid):
    singlerow = []
    for row in grid:
        singlerow.extend([x is None and -1 or (x is not None and x)
        for x in row])
    return tuple(singlerow)

def gridString(grid):
    return "\n".join([" ".join([x is None and "_" or str(x)
    for x in row])
    for row in grid])

if __name__ == "__main__":
    master()
empty = sum([len([x for x in row if x is None]) for row in grid])

if not linda.connect():
    print "Please start the Linda server first."
    return

start = time.time()
linda.uts._out((empty, ) + gridToTup(grid))

tup = linda.uts._in((0, ) + (int, ) * (9*9))

print "Solved Grid in %f seconds." % (time.time() - start)
print gridString(tupToGrid(tup[1:]就来看看)

def loadFile(filename):
    fp = open(filename, "r")
    rows = []
    for line in fp:
        if len(rows) == 9:
            break
        cols = line.strip().split(" ")
        if len(cols) != 9:
            raise SystemError, "Row (%s) only has %i columns." % (line, len(cols))
        cols = [(x != "_" and int(x)) or (x == "_" and None) for x in cols]
        rows.append(tuple(cols))
    if len(rows) != 9:
        raise SystemError, "File only has %i rows." % (len(rows), )
    return tuple(rows)

def tupToGrid(tup):
    assert len(tup) == 81

    grid = []
    for r in range(9):
        grid.append([(x != -1 and x) or (x == -1 and None) for x in tup[r*9:(r+1)*9]])
    return grid

def gridToTup(grid):
    singlerow = []
    for row in grid:
        singlerow.extend([(x is None and -1) or (x is not None and x) for x in row])
    return tuple(singlerow)

def gridString(grid):
    return "\n".join([" ".join([(x is None and "_") or str(x) for x in row]) for row in grid])

231
if __name__ == "__main__":
    master()

D.2.2 Compound Types

Master Process

#!/usr/bin/python

import sys
import time

import linda

if not linda.connect():
    print "Please start the Linda server first."
    sys.exit(1)

class Grid:
    def __init__(self):
        self.grid = tuple(tuple([None for _ in range(9)]) for _ in range(9))

def gridToValue(grid, memo):
    return linda.Value(grid.grid)

def valueToGrid(val, memo):
    g = Grid()
    for x in range(9):
        for y in range(9):
            g.grid[x][y] = val[x][y]
    return g

gridtype = linda.Type("grid :: ((int + Nil) * 9) * 9;", Grid,
                        gridToValue, valueToGrid)

def master():
    grid = loadFile(sys.argv[1])

    print "Solving..."
    print gridString(grid)

    empty = sum([len([x for x in row if x is None]) for row in grid.grid])

    start = time.time()
    linda.uts._out((empty, grid))

    tup = linda.uts._in((0, gridtype))

    print tup[0]
assert tup[0] == 0, tup[0]

print "Solved Grid in %f seconds." % (time.time() - start)
print gridString(tup[1])

def loadFile(filename):
    fp = open(filename, "r")
    rows = []
    for line in fp:
        if len(rows) == 9:
            break
        cols = line.strip().split(" ")
        if len(cols) != 9:
            raise SystemError, "Row (%s) only has %i columns." %
            (line, len(cols))
        cols = [(x != "_" and int(x)) or (x == "_" and None) for x in cols]
        rows.append(tuple(cols))
    if len(rows) != 9:
        raise SystemError, "File only has %i rows." % (len(rows), )
    g = Grid()
    g.grid = tuple(rows)
    return g

def gridString(grid):
    return \\
    "\n".join([" ".join(["x is None and ""] or str(x)
        for x in row])
        for row in grid.grid])

if __name__ == "__main__":
    master()

Slave Process

#!/usr/bin/python

import sys
import time

import linda

def master():
    grid = loadFile(sys.argv[1])

    print "Solving..."
    print gridString(grid)

    empty = sum([len([x for x in row if x is None]) for row in grid])

    if not linda.connect():
        print "Please start the Linda server first."
        return
start = time.time()
linda.uts._out((empty, ) + gridToTup(grid))

tup = linda.uts._in((0, ) + (int, ) * (9*9))

print "Solved Grid in %f seconds." % (time.time() - start)
print gridString(tupToGrid(tup[1:]))

def loadFile(filename):
    fp = open(filename, "r")
    rows = []
    for line in fp:
        if len(rows) == 9:
            break
        cols = line.strip().split(" ")
        if len(cols) != 9:
            raise SystemError, "Row (%s) only has %i columns."
            % (line, len(cols))
        cols = [(x != "_" and int(x)) or (x == "_" and None)
            for x in cols]
        rows.append(tuple(cols))
    if len(rows) != 9:
        raise SystemError, "File only has %i rows."
        % (len(rows),
    return tuple(rows)

def tupToGrid(tup):
    assert len(tup) == 81

    grid = []
    for r in range(9):
        grid.append([x != -1 and x) or (x == -1 and None)
            for x in tup[(r*9):(r+1)*9]])
    return grid

def gridToTup(grid):
    singlerow = []
    for row in grid:
        singlerow.extend([(x is None and -1) or (x is not None and x)
            for x in row])
    return tuple(singlerow)

def gridString(grid):
    return "\n".join([" " .join([str(x)
        for x in row]) for row in grid])

if __name__ == "__main__":
    master()
D.3 Sudoku Generator

This section contains the complete code for the case study presented in section 7.3.

D.3.1 Master Process

#!/usr/bin/python

import sys
import time
import linda

if not linda.connect():
    print "Please start the Linda server first."
    sys.exit(1)

class Grid:
    def __init__(self):
        self.grid = tuple([tuple([None for _ in range(9)]) for _ in range(9)])

def gridToValue(grid, memo):
    rows = []
    for row in grid.grid:
        cols = []
        for col in row:
            if col is None:
                v = linda.Nil()
                v.sum_pos = 1
            else:
                v = linda.Value(col)
                v.sum_pos = 0
            cols.append(v)
        rows.append(tuple(cols))
    return linda.Value(tuple(rows))

def valueToGrid(val, memo):
    g = Grid()
    for x in range(9):
        for y in range(9):
            if val[x][y].sum_pos == 1:
                v = linda.Nil()
                v.sum_pos = 1
                g.grid[x][y] = v
    return g

gridtype = linda.Type("grid :: ((int + Nil) * 9) * 9;", Grid,
                        gridToValue, valueToGrid)

def master():
grid = Grid() # Create an empty grid

work_ts = linda.TupleSpace()
linda.uts._out(("work_ts", work_ts))

func = linda.loadCodeFromFile("difficulty_func.min")
linda.uts._out(("difficulty_func", func))

start = time.time()
linda.uts._out((empty, grid))

tup = linda.uts._in((gridtype, ))

assert tup[0] == 0, tup[0]

print "Generated Grid in %.10f seconds." % (time.time() - start)
print gridString(tup[1])

def gridString(grid):
    return "\n".join(" "
        .join([x is None and "_" or str(x)
            for x in row]) for row in grid.grid])

if __name__ == "__main__":
    master()

D.3.2 Generator Process

#!/usr/bin/python

import sys
import time
import copy

import linda

if not linda.connect():
    print "Please start the Linda server first."
    sys.exit(1)

class Square:
    def __init__(self, val):
        self.val = val

    def squareToValue(sq, memo):
        if sq.val is None:
            v = linda.Nil()
            v.sum_pos = 0
        else:
            v = linda.Value(sq.val)
            v.sum_pos = 1
        return v
def valueToSquare(val, memo):
    if val.isNil():
        return Square(None)
    else:
        return Square(int(val))

sqtype = linda.Type("square :: Nil + int;", Square, squareToValue, valueToSquare)

class Row:
    def __init__(self, row):
        self.row = row

def rowToValue(row, memo):
    return linda.Value(row.row)

def valueToRow(val, memo):
    return Row(list(val))

rowtype = linda.Type("row :: square * 9;", Row, rowToValue, valueToRow)

class Grid:
    def __init__(self, rows):
        self.rows = rows

def fillInSquare(self):
    for x in range(9):
        for y in range(9):
            if self[(x,y)] is None:
                newgrids = []
                for v in self.getValid(x, y):
                    newgrids.append(copy.deepcopy(self))
                    newgrids[-1].rows[y].row[x] = v
                return newgrids
    return []

def getValid(self, x, y):
    possible = {1: True, 2: True, 3:True, 4: True: 5: True
6: True, 7: True, 8:True, 9:True, None: False}
    for ny in range(9):
        if ny != y:
            possible[self[(x, ny)]] = False
    for nx in range(9):
        if nx != x:
            possible[self[(nx, y)]] = False
    for nx in range(3*(x/3), 3*(x/3)+3):
        for ny in range(3*(y/3), 3*(y/3)+3):
            if nx != x and ny != y:
                possible[self[(nx, ny)]] = False
    return [x for x in possible.keys() if possible[x]]
def __getitem__(self, (x, y)):
    return self.rows[y].row[x]

def gridToValue(grid, memo):
    return linda.Value(grid.rows)

def valueToGrid(val, memo):
    return Grid([row for row in val])

gridtype = linda.Type("grid :: row * 9;", Grid, gridToValue,
valueToGrid)

def generator():
    _, work_ts = linda.uts._out(("work_ts", linda.TupleSpace)
    while True:
        empty, grid = linda.uts._in((int, gridtype))

        newgrids = grid.fillInSquare()

        for g in newgrids:
            work_ts._out((empty-1, g))

if __name__ == "__main__":
    generator()

D.3.3 Solver Process

#include "linda.h"

int main(int argc, char* argv) {
    Linda_connect();

    LindaValue grid = Linda_type("grid :: ((Nil + int) * 9) * 9;";
    LindaValue functype = Linda_type("difficulty :: grid -> bool;";

    LindaValue template = Linda_tuple(2);
    Linda_setTuple(template, 0, Linda_string("difficulty_func"));
    Linda_setTuple(template, 1, functype);

    LindaValue tup = Linda_rd(Linda_uts, template);
    LindaValue difficulty_func = Linda_getTuple(tup, 0);
    Linda_addReference(difficulty_func);
    Linda_delReference(tup);

    Linda_setTuple(template, 0, Linda_string("work_ts"));
    Linda_setTuple(template, 1, Linda_tupleSpaceType);

    LindaValue tup = Linda_rd(Linda_uts, template);
    LindaValue work_ts = Linda_getTuple(tup, 0);
    Linda_addReference(work_ts);
    Linda_delReference(tup);
Linda_delReference(template);
template = Linda_tuple(1);
Linda_setTuple(template, 0, grid);
while(1) {
    LindaValue args;
    LindaValue r;

    tup = Linda_inp(work_ts, template);
    if(tup == NULL) {
        break;
    }

    args = Linda_tuple(1);
    Linda_addReference(Linda_getTuple(tup, 0));
    Linda_setTuple(args, 0, Linda_getTuple(tup, 0));

    r = Linda_callFunction(difficulty_func, args);
    Linda_delReference(args);
    if(Linda_isTrue(r)) {
        args = Linda_tuple(2);
        Linda_setTuple(template, 0, Linda_string("sudoku"));
        Linda_addReference(Linda_getTuple(tup, 0));
        Linda_setTuple(template, 1, Linda_getTuple(tup, 0));
        Linda_out(Linda_uts, args);
    }
    Linda_delReference(tup);
}

D.3.4 Difficulty Function

Grid :: ((int + Nil) * 9) * 9;

# We're looking for a grid which has 60 numbers.
# This almost certainly won't be a proper Sudoku grid, but
# it'll look like one...
difficulty :: Grid -> bool;
difficulty grid = count_numbers(grid) == 60;

# Loop through the grid counting the numbers
count_loop :: Grid -> int -> int -> int;
count_loop grid x y =
    if x != 8 and y != 8
    then
        if grid[x][y] == Nil then 0 else 1 endif
    else
        if grid[x][y] == Nil then
            count_loop(grid, (if y == 8 then x + 1 else x), (y+1)%9)
        else count_loop(grid, (if y == 8 then x + 1 else x), (y+1)%9)
            + 1
        endif

239
# Call count_loop to loop through the grid

count_numbers :: Grid -> bool;
count_numbers grid = count_loop(grid, 0, 0);
Bibliography


