Semantic lifting and active libraries

Transfer Report

Peter Collingbourne

September 2008

Abstract

An active library is a software library that can optimise itself to its environment. This report describes *semantic lifting*, essentially the extraction of high-level semantic information from a low-level process description by means of program analysis, and discusses how we may use this idea to extend the concept of active libraries to include program analysis and verification capabilities. We give two case studies that show our preliminary research results in this field and discuss how these ideas may be extended to continue our research.

Contents

Chapter 1

Introduction

This report describes the research carried out during the first twelve months of my research degree. The purpose of this chapter is to explain my general research ideas with reference to the later chapters of this report which describe the ideas in more detail.

1.1 Motivation and Outline

In the course of software library design we may encounter situations where a library must, due to the nature of its design, carry out a transformation step or impose certain constraints on the calling program. An example of a program transformation may be an optimising transformation specifically tailored to the operation of the library. Examples of constraints we may wish to impose include properties such as safety and liveness, which in turn may lead us to more sophisticated analyses. The former situation falls under the widely studied concept of *active libraries* [91]. The goal of this work is to examine techniques for augmenting libraries and compilers in order to expand the definition of active libraries to encompass both advanced transformation and analysis behaviour.

We begin our study by introducing the concept of semantic lifting, which, in essence, is the extraction of high-level semantic information from a low-level process description by means of program analysis. Semantic lifting is the key to implementation of active libraries which incorporate semantic information into their analysis and transformation processes. Semantic lifting depends on the availability of tools which allow us to parse program code written in standard programming languages (in compiler terminology, these are known as "front-ends"), and the existence of program analyses that can extract useful information from a program. An example of such an analysis may be a control flow analysis which works with programs that use libraries that may be modelled as a finite state automaton, in order to check that the control flow graph reduces to the automaton. From a research perspective, the most important detail is the analysis process, so the central research challenge is to carefully develop anal-

yses that we can use to prove interesting program properties, and incorporate them into our active libraries.

The work covers a range of concepts in the fields of theory and practice, among which are the following:

- Program analysis is an important keystone for semantic lifting, as it provides the technique by which information is extracted from a program.
- Session typing, an established theoretical concept, is given particular prominence in this work primarily for the constraints it imposes on programs that use it, which we can verify using program analysis techniques.
- Meta-object protocols provide a mechanism for extracting information from a program. They also provide a mechanism for modifying a program, possibly to reflect information thus extracted.

These concepts and more are further discussed in Chapter 2.

We believe that the current body of research is incomplete in the context of the scope of this work, and more research is required in order to consider the application of semantic lifting in combination with active libraries. This point is further considered in Chapter 3.

We have two case studies which illustrate the research done up to this point. The first case study, discussed in Chapter 4, covers an example of an active library that implements the concept of session typing which has been developed as part of this research, in which certain instances of semantic lifting are necessary to implement the library's restrictions. The second case study, discussed in Chapter 5, covers preliminary research into developing advanced metaprogramming support for the C++ language by augmenting its template metaprogramming mechanism with a more powerful functional language, in which we have implemented an example of a concept which would be hard if not impossible to implement using C++ template metaprogramming.

I plan to continue my research by completing the extended metaprogramming support for C++ thus allowing for further results to be made available. The research shall then proceed by considering and implementing a number of applications of the metaprogramming extension and considering the impact of adding similar capabilities to other programming languages. The plans are covered in further detail in Chapter 6.

1.2 Summary

We have briefly described the current research in the field, remarked upon the deficiencies therein and introduced our preliminary research. We have also provided references to later chapters in the report which describe these points in a greater amount of detail.

Chapter 2

Background

This chapter introduces some of the key background information involved in this research, including a comprehensive review of the important concepts underlying this work. We also look at some of the specific concepts involved in some of the preliminary research in this area, with a view to their applicability in the context of this research. While this chapter provides an introduction to the underlying concepts, Chapter 3 details our literature survey of the main prior research in the field.

2.1 Active Libraries

An *active library* is an augmented instance of a traditional software library (that is, a collection of functions and classes) that takes an active role in the compilation of a program in order to specialise the implementation to the calling program depending on its requirements. This role may include a specialised form of whole-program optimisation, the generation of additional components or providing additional information to the runtime environment such as specialised debugging information. An important issue in the development of active libraries is whether or not a particular feature should be incorporated into the compiler or into the library itself. Where the language provides facilities which allow for the development of such features, such as the C++ language's template metaprogramming facilities, it is usually preferable to utilise these in order to minimise complexity. Where the facilities must be provided outside of the language, or from a complexity perspective it would be easier to provide them outside of the language, there are generally three approaches one may take.

The first approach is to use a domain-specific language or DSL. One can consider DSLs to exist orthogonally to active libraries – whereas active libraries usually exist as a superset of the base language, a DSL exists as a separate language with its own syntax and semantics. DSLs may be deployed for a number of reasons including improving the notation from that provided by the host

language, and freeing the programmer from the burden of writing "boilerplate" code.

The second approach is to use an external tool to provide either program transformation or verification capabilities. This approach allows for a clean separation of concerns; however, depending on the requirements of the library in question, the tool in question may need to act as a full front-end for the development language. For some languages, such as LISP and variants, this is trivial due to the ease with which the language can be parsed and the fact that the language has a standard representation of its program code as a data structure in that language. Certain other languages, such as C++, are notoriously hard to parse correctly, requiring the investment of many years of development time. Even now, only a handful of front-ends exist for C++, including the GNU Compiler Collection's front-end and the EDG front-end.

The third approach depends on being able to augment the compiler with any necessary facilities required to perform such translation or analysis. This approach requires considerable knowledge of the internals of the compiler and depends on the compiler's internal data structures being organised in an easily manipulable manner. In some cases, this may be considered easier than writing a front-end from scratch.

2.2 Semantic Lifting

A suite of programming tools may consist of a low-level imperative language, a high level logical process description, and a set of tools for analysing and manipulating both. Imperative languages are chosen for their ability to express a number of real-world programming problems, however they rarely contain facilities for expressing provable properties, such as safety and correctness, for a particular program. On the other hand, higher-level logical descriptions provide more scope for deductions to be made. In this work we consider semantic lifting to be the process of transforming low-level process descriptions into high-level descriptions for analysis. This process of transformation will generally be carried out using various program analysis techniques.

The information extracted using semantic lifting may be highly useful to an active library. For example, we may wish to use control flow information extracted in order to statically verify that communication over a session with an associated session type conforms to the sequence of communication actions stipulated in the session type.

A related concept is *semantic hinting*. Semantic hinting occurs when the semantic information is not inferred but is contained within the program, for example by means of a pragma or type annotation. Semantic hints may indeed be the end-product of semantic lifting, if the tool operates in a program transformation capacity.

2.3 Program Analysis

The static analysis of computer programs is a prominent topic which spans both theory and practice. Frequently when carrying out program analysis, we take account of the following attributes of the program:

- The semantics of the program statements;
- The control flow information extracted from the program.

Using these attributes a wide range of analyses may be performed. We may wish to carry out a data flow analysis, in order to determine how data is manipulated within the program. A classical data flow analysis is the live-variable analysis which determines, for each program point, which variables may possibly used at a later stage in the program. Most data flow analyses use an iterative approach to propagate analysis information throughout the program until it reaches a fixed point, or stabilises.

2.4 Abstract Interpretation

A common program analysis technique is known as abstract interpretation [28, 30]. One may characterise an abstract interpretation as a generalisation or *abstraction* of the state of a computer program, perhaps expressed in more specific terms, such as an abstraction of the value of a particular variable. Abstract interpretations are designed in such a way that they may be determined statically using control and data flow analyses.

An example of an abstract interpretation is one over the set of integers that is used to determine the sign of the number statically. We do this by defining the set of values in our abstraction:

$$\{\bot, -1, 0, +1, \top\}$$

Here -1 represents a negative number, +1 a positive number, 0 the number zero, $\top$ an unknown value and $\bot$ an inconsistency. We also define an *abstraction function* that converts concrete values in the set of integers to abstract values in our abstraction:

$$f(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ +1, & x > 0 \end{cases}$$

We may then assign abstract semantics to mathematical operations such as addition and multiplication. Here we give an example for the addition operator:

$+$	$\bot$	-1	0	$+1$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
-1	$\bot$	-1	-1	$\top$	$\top$
0	$\bot$	-1	0	$+1$	$\top$
$+1$	$\bot$	$\top$	$+1$	$+1$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

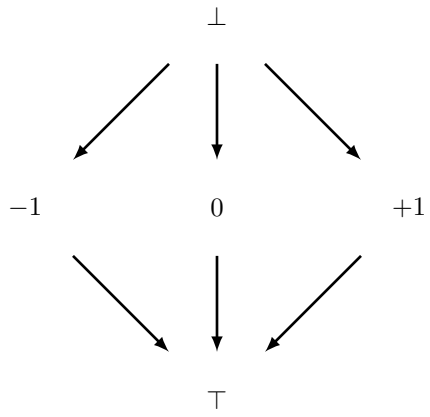

Figure 2.1: Partial order for our abstract interpretation. Arrows indicate the $\leq$ relation.

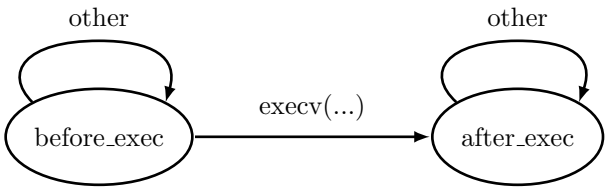

Figure 2.2: Model of the `exec` system call

When multiple control flow paths converge, we need to combine our results for each path. The function that combines the results together is known as the join function. In simpler cases, the join function is defined as the least upper bound of the set of results using a partial order over the set of values in the abstraction. In our case the partial order is defined as in Figure 2.1.

We can now apply our abstraction function to any constant value assigned to the variable, and abstract semantics to any computation carried out that is assigned to the variable. The result is that at each program point we know the variable's value in terms of the abstraction. In this case, we shall know the sign of each integer variable's value, where that information is available.

There exist a number of nuances associated with the application of abstract interpretation, which are discussed in more detail in Chapter 3.

2.5 Model Checking

Abstract interpretation has inspired the field of model checking [25, 24, 80], essentially an abstraction over program semantics rather than data. The goal

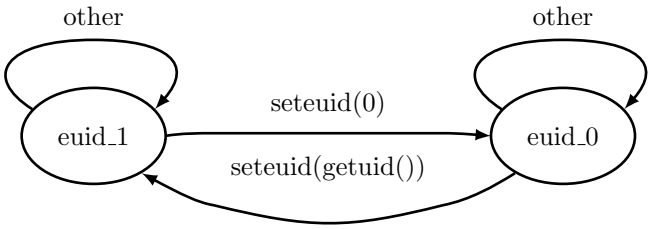

Figure 2.3: Model of the `seteuid` system call

of model checking is to verify that a program satisfies a *model* or specification. An example of such a specification may be a finite-state automaton. For example, we may wish to verify the security property that a `setuid` program does not give superuser privileges to a program it executes (example from Chen and Wagner [20]). We model the system calls `exec` and `seteuid` using the finite-state automata shown in Figures 2.2 and 2.3. We then verify that we cannot enter states `after_exec` and `euid_0` simultaneously, by reducing our program to the product of the two automata. If there is no combined state (`after_exec`,`euid_0`), we conclude that the program cannot enter this state.

2.6 Pluggable Type Systems

Pluggable type systems [16] are a mechanism for layering type systems onto an initially untyped language. Their appeal in this work comes about due to the programmatic nature of the type system. The inheritance structure, semantics and other properties may all be defined programmatically. Such type systems may also impose semantic constraints on variables with those types (e.g. on modification, access, or some domain specific constraint). We may also wish to infer the specific pluggable types that may be assigned to a variable. The inference process may be considered a form of semantic lifting, as the inferred types may reflect the semantic properties of how that variable is used. Or, if the pluggable type system simply plays the rôle of a verifier without capability for inference, one my consider these type systems to be a form of semantic hinting.

Pluggable types may be used in a variety of ways. For example, the types inferred by the plugins may provide information to the compiler that aids optimisation by making certain information, such as bit widths or structure layouts, available at compile time.

2.7 Session Types

The session type [54] is a means of characterising dyadic interaction between processes over a communication channel. A session type is a property of a session, a communication link established over a channel. Process interactions

are expressed as a sequence of communication actions, and any communication taking place over the session with which the type is associated must conform to the sequence of actions. Although the roots of session typing can be traced to the π-calculus [70], it has also been applied to a wide range of programming paradigms, including object-oriented imperative programming [34].

In most existing literature, session types are expressed as expressions with a specified syntax. Session type syntax is generally recursive. This allows for arbitrary composition of communication actions in whichever form fits the structure of the program. A session type may take a number of forms, whose semantics we shall briefly describe. A session type may be an *action*. An action specifies a communication direction (in or out) and type (this may be a language-specific primitive type or, in the case of delegation [34], another session type). An action represents, dependent on the communication direction, the reception or transmission of a value (or session) of the specified type. A session type may also be the *sequential composition* of two or more session types. The session type that is the composition of one or more session s_1, $s_2 ... s_n$ represents the actions in s_1, followed sequentially by those in s_2 and so on up to s_n. A session type may also be a *choice* between a number of sessions s_1, $s_2 ... s_n$. A session type may also be the *terminating session type*. A session with the terminating session type may not perform any communication actions or change its type. It may only close the communication channel.

When implementing session typing within a language, we frequently wish to impose constraints upon the program code which uses it. Using an appropriate program analysis tool, we may determine if a program satisfies these constraints.

2.8 The GNU Compiler Collection

The GNU Compiler Collection, or GCC, is one of the most popular compilers in use today. It supports a wide range of programming languages, including C, C++, Objective C, FORTRAN 77 and Java, and a wide range of computer architectures. GCC is notable due to its open source nature, meaning that anyone may inspect and modify the source code. For some languages, GCC remains the highest quality open source compiler available today.

2.9 Metaprogramming

Metaprogramming, or "programming about programming", refers to various techniques for writing programs that, rather than manipulating data, manipulate programs. The nature of this manipulation may be indirect (e.g. compile-time computation of a value or type to be used elsewhere in the program), or more "hands-on" (e.g. building an Abstract Syntax Tree from scratch). Metaprogramming is one of the primary techniques that can be used in the construction of active libraries.

One of the first languages to support meta programming was LISP, which

has a native representation of itself as a data structure, and a special variant of a function known as a *macro*, which instead of returning a value returns a program in the aforementioned data structure. The program is then substituted into the place where the macro was called.

More recent metaprogramming languages have emerged, such as Template Haskell, which is an extension to Haskell that allows generation of function definitions at compile time by creating an abstract syntax tree. In this way it is similar to Lisp macros, but because Haskell is a strictly typed language, type checking is performed when a template is expanded.

The C++ template metaprogramming mechanism is a powerful compile-time technique which relies on various obscurities in the technical details of C++ template instantiation in order to provide a Turing-complete [89] computation mechanism. There exist libraries which expose highly complex compile-time operations (such as sorting) which are written entirely using template metaprogramming, a prominent example being the Boost Meta-Programming Library (MPL).

2.10 Meta-Object Protocols

Meta-object protocols are a particular type of metaprogramming, which provide an object-oriented interface to a programming language, at a meta-level. They also provide the mechanism by which one may extend the syntax of a language (for example by adding a new domain-specific loop statement, or by adding type annotations as part of a pluggable type system). These meta-level objects represent parts of the language, such as the compiler's intermediate representation and its mechanism for interpreting certain operations (such as adding two objects together). Typically, the meta-level objects can be manipulated in the language itself, but sometimes a DSL is used. These meta-level objects may be extended in order to modify the behaviour of the language. This is achieved by providing a transformation capability to the metaclass, by which it may change or completely substitute the default behaviour of the language.

Meta-object protocols may operate at the compile-time level or the run-time level. These levels denote at which stage the metaobjects are used by the language. For compile-time MOPs, metaobjects typically operate at the intermediate representation level in that meta-operations are carried out over IR nodes. For runtime MOPs, metaobjects may have access to the AST (depending on the language), but most also have access to the program's run-time state and therefore can participate directly in the operation of the program.

At the compile-time level, MOPs can act as an enabling tool for active libraries, as the active library may implement its own metaclass and thereby take an active role in the compilation. MOPs also enable semantic lifting, as the information provided via the metaclasses may be analysed for semantic information, for example by using a control flow analysis.

2.11 Theorem Provers

An automated *theorem prover* [18] is a program that finds a proof of a result (only if one exists) given a conjecture. Theorem provers have been used for a variety of purposes including program analysis. In the context of program analysis, theorem provers may be used to prove high-level program properties by deriving them from low-level properties about, for example, individual program statements. In this way, theorem provers carry out a form of semantic lifting.

2.12 Ownership Types

In the context of program verification, it is always useful to be able to soundly restrict the scope of an analysis in order to streamline the process – this is known as *local reasoning*. Due to the heavy use of aliasing within object-oriented programming languages as compared to conventional languages, it is often difficult to carry out a sound program verification, as we must consider every possible use of an object. The *ownership types* [23] mechanism describes how we may impose a constraint on an object-oriented programming language in order to mitigate the negative effects of aliasing with respect to program verification, by restricting the set of objects which may have direct access to a particular object. The ownership type mechanism has been designed as a type system that may be verified statically at compile time.

Under ownership typing, each object is assigned an *owner* which is generally the only object that may access the object, apart from the object itself and certain other objects. By restricting access in this way, we may soundly restrict our program analysis to the set of objects which have a meaningful relationship with the object.

The concept of ownership types acts as a form of semantic hinting, as it denotes within the program text those objects with a semantic relationship between each other. It also achieves the active library goal of greater efficiency, as it reduces the required scope of a particular program analysis. However, we must not be confused by the terminology: ownership types do not constitute hints as such, rather they exist as explicit statements of the relationships between object.

2.13 Summary

In this chapter we have described the concepts related to this research, and explained how they relate together in order to provide a cohesive body of concepts to build research on. In particular we have looked at various program analysis techniques, such as model checking and abstract interpretation, which provide a foundation for carrying out semantic lifting and hinting. We have also looked at a number of enabling mechanisms for our work, namely metaprogramming and meta-object protocols. We have also looked at areas where our work may be used effectively, such as session typing, which is a prominent example of a

programming paradigm that must use program analysis techniques in order to verify the constraints imposed on the program.

Chapter 3

Related Work

This chapter presents the related work in the field and builds a case for future research. This is achieved by citing a number of examples of applications of our concepts from the literature. For our new concepts, of course this work may not reference them by name, but the provision of a number of examples will demonstrate the usefulness of naming the concepts, and paint a clearer picture as to the precise scope of these concepts.

3.1 Active Libraries

Although active libraries were first introduced in 1998 by Veldhuizen et al [91], one could say that active libraries have many years of history behind them. Although they were not considered that way at the time, the introduction of metaprogramming in the form of macros in the LISP language and and its variants provided an implementation technique that could be used to implement active libraries. The macro as we know it today was perhaps first introduced by McIlroy in 1960 [69], which describes how metaprogramming capabilities could be introduced into an Algol compiler.

After macros were added to LISP [83], an interesting extension to the language was proposed by Kohlbecker and Wand [63] which provided a form of pattern matching implemented via macros. This work later became a part of the standard for the LISP variant Scheme in 1991 [1]. As a further example, Krishnamurthi [64] gives an example of an automata language implemented in Scheme. These extensions could be considered relevant here because they are examples of a concept we can observe in the field of active libraries, namely that of semantic information being used to generate additional specialised code (namely the pattern matching or automata code).

The Veldhuizen paper cites a number of examples of active libraries. Among these is the scientific computational library, Blitz++ [90], which avoids the common inefficiencies encountered when performing computations in C++ (due to the large number of intermediate values produced) by using the technique of

expression templates to collapse all computation into a single loop.

Further examples of active libraries include De Sutter et al [85], which describes a mechanism for creating specialisable libraries, which may be considered active libraries within the Veldhuizen definition. Libraries may be specialised using the subset of methods that are used by the caller. This leads to optimisation in that we may choose an implementation suited to a particular task (for example, a random access list), or an implementation that does not require certain features (such as multiprocessing capabilities).

Lemeur et al [65] present their system for scenario-specific specialisation, as a component-based extension of the C language. *Specialisation scenarios* are defined, which describe how modules (components) may be specialised, by setting out the specific conditions for specialisation. For instance, we may specialise a matrix multiplication routine to the sizes of the arrays. In this way, the modules act as active libraries within the Veldhuizen definition.

Guyer and Lin [48] is a particularly apt example in the context of this thesis. It describes a compiler known as Broadway which allows for augmentation of system calls with semantic information relating to its operations. The system libraries, taken together with the augmentations, represent an active library, as the augmentations describe how optimisations may take place. The augmentations themselves represent a form of semantic hinting. Optimisations may include inlining, or outright replacement of a particular code fragment with another, depending on the particular circumstances, which can be tailored to the particular library routine. The augmentations also allow for a certain amount of program verification, with the capability for custom error messages in the case where verification fails. An example given by the authors is a solver routine for the mathematical library PLAPACK [3]. An *action annotation* is given which provides that in the case where the routine is called in the case where the input matrices are empty, a fact that can sometimes be determined statically, the procedure call is removed.

One may also consider systems such as Stratego [93]. Under this system, not only can the user define rewrite rules that describe transforming optimisations to the code, the user may also give *strategies* for traversing the intermediate representation. We can consider that the rewrite rules taken together with a particular library form an active library, and the strategies together with rewrite rules give the system semantic guidance on the structure of the program. A related work is Sittampalam et al [82]. In this system, programs and transformation operations are represented by Prolog relations, and a transformation specification also has access to control flow information which is presented as *annotations* to the schema.

3.2 Pluggable Type Systems

The existing work on pluggable type systems covers both theory and implementation in a number of languages. The paper by Bracha [16] introduces the concept. The central idea of this paper is that there may exist a number of

type systems for various purposes, each constituting a "plugin" to the compiler. One may also consider that multiple interacting analysis plugins can sometimes give more precise information than a single plugin operating independently. This concept was arguably introduced, in a more general sense, by Cousot and Cousot [29], who show that combining program analysis frameworks lead to more accurate information than the conjunction of the corresponding separate analyses.

Charlton [19] describes the theoretical basis for these type systems with some discussion of combining type systems for preciseness. Foster et al [42] presents type annotations an a lattice framework, and embraces the work of Dussart et al [49] by treating the binding-time analysis of Jones and Muchnick [60] as such an annotation.

The idea has been expanded upon in later work, with implementations and applications described. Due to the flexible attribute syntax for Java [36], the implementation of pluggable type systems for this particular language has been extensive. Andreae et al [5] introduce a DSL for describing pluggable type systems in Java. The language, known as JavaCOP, allows one to impose rules on particular program elements. These rules may involve references to the types of the expressions contained within the elements, including any attributes which may exist on these types. In this way, we may constrain the use of values with type annotations.

Another similar work is Papi et al [74]. This work takes an alternative approach in that the pluggable type systems are implemented in Java rather than a DSL. Checkers have access to an Abstract Syntax Tree from which annotation information can be extracted.

The above cited works have been used to implement statically verified constraints for the Java attribute `@NonNull`, first introduced by Fähndrich and Leino [37], which imposes the constraint on any variable with this attribute that the variable may never contain the value `null`. Both works have also been used to implement the Javari [87] reference immutability constraint for Java. Reference immutability of a reference variable imposes a read-only constraint on the object to which the variable refers, similar to the `const T*` construction in C++.

Further work on pluggable type systems has been carried out for other languages. For Haskell, Xu [98] presents an implementation of *contracts* that frames each contract as a set of type annotations that act as pre- and postconditions for a function. For C♯, the Spec♯ [11] language provides a number of contributions, among which is a non-null type system.

Pluggable type systems have also made their way into C and C++, but in a limited way. Foster et al [42] present implementation work on `const`ness inference and polymorphic inference for C and discuss a number of other annotations. Collingbourne [27] describes an implementation of the linearity constraint for C++, a constraint which if imposed upon a variable prevents the variable from being used (read) more than once after it has been defined (set), however this implementation operates as a checker outside of the context of the compiler.

3.3 Abstract Interpretation

Starting from the Cousot paper [28], the subject of abstract interpretation has received a great deal of study in a variety of different contexts. We shall attempt to highlight those works which have particular relevance here.

Wilson and Lam [97] describe their system for carrying out pointer analysis of C programs, implemented for the SUIF [96] compiler framework. By necessity, this system operates interprocedurally due to the procedural nature of that particular programming language. In their system, an analysis of a particular procedure uses information about which alias relationships hold at the entry point of that procedure. The alias relationship information acts as an abstraction of the full calling context of the procedure call. Consequently, their analysis is made more efficient.

In the field of security, Wagner et al [95] present their approach to detecting buffer overflows in C code by employing an integer range interpretation over the set of integers, and a data size abstraction over the set of string pointers. Another security related topic is information flow. Barbuti et al [10] show how one may use an analysis of a subset of Java bytecode to certify that no information at a certain security level flows to a lower level.

Ferdinand et al [38] describe their system for approximating the execution time of a program while taking into account the status of the cache. The cache is represented as a series of abstract cache lines, which indicate which memory regions may be contained within a particular line.

3.4 Model Checking

Model checking was first introduced by Clarke et al [25, 24] as a viable program analysis technique. In keeping with the focus of this thesis, we shall now examine some uses of model checking in practical contexts.

SPIN [53] is one of the most prominent model checkers in use today. SPIN verifies LTL [77] properties of program models written in a language known as Promela [51]. SPIN utilises a number of techniques, such as partial order reduction [75], state compression and bit-state hashing [50] in order to optimise the verification process. Another tool, AX [52] has been developed for automatically extracting a Promela model from a C program for use with SPIN, thus permitting the verification of existing C programs.

MOPS [20] is a finite-state model checker for the C programming language, written in Java. In MOPS, we model a property of a system that we wish to verify as a finite state automaton, where any undesirable state (such as a state which would create a security vulnerability) is marked as unsafe. Transitions through the automaton are represented by particular statements encountered within the program. The program is converted to a pushdown automata (PDA), and composed with the system model FSA. A reachability analysis is then used to determine if an unsafe state is reachable, and if so a trace is output.

Microsoft's SLAM [9] toolkit, commercially the Static Driver Verifier, is used

to statically verify Windows NT device drivers written in C against certain safety properties written in a specification language known as SLIC. SLIC acts as a semantic specification for a number of Windows kernel system calls, and describes these system calls in abstract terms. SLAM operates using three tools which are applied iteratively in order to obtain a more precise result: C2bp, Bebop and Newton. C2bp is introduced by Ball et al [6] as a tool for extracting a boolean program [7] from a C program, using predicate abstraction as introduced by Graf and Saïdi [47] and a set of given predicates. Bebop [8] carries out the actual model checking task on the boolean program generated by C2bp.

3.5 Meta-Object Protocols

A number of meta-object protocols have been implemented, for a variety of languages. The Common Lisp Object System's meta-object protocol [61] was one of the first such protocols.

Ishikawa et al [59] present their work on a meta-level architecture for C++ known as MPC++ which exposes the meta-level architecture using a C++ model, and allows the user to create new type modifiers, declarator modifiers, expressions, statements and structural declarations simply by extending the relevant classes of the C++ representation. Chiba [21] presents a similar architecture known as OpenC++ which allows the user to define custom modifiers, access specifiers, while-style statements, for-style statements and closure statements. OpenC++ is distinct from MPC++ in that each class is associated with a metaclass, which defines translation functions that define the transformation strategy for that class, whereas MPC++ transformations operate globally. Translation functions operate on particular syntax elements associated with a class, such as member function calls, data member access and the class declaration.

We may also consider the potential for a meta-object protocol afforded by Java. The language provides a number of tools for building a runtime meta-object protocol. Firstly, a basic MOP is provided in the form of its *reflection* mechanism, however this mechanism is restricted in that one cannot extend the reflection objects, only introspect them. Secondly, the language provides support for runtime loading of bytecode, which allows for more sophisticated MOPs to be implemented. For example, the Javassist tool of Chiba [22] provides support for both compile-time and runtime meta-object protocols, using the bytecode generation technique.

Meta-object protocols have also been implemented for Java at the compile-time level. Tatsubori et al [86] present their OpenJava system, which uses OpenC++'s metaclass technique and allows for new class/member modifiers and clauses to be introduced.

Meta-object protocols have been used in a variety of ways. Böllert [13] describes his work on a dynamic weaver for aspect-oriented programming [62] known as the AOP/ST Weaver. It operates using Smalltalk's runtime MOP, a

native reflection mechanism [40].

3.6 Session Types

Session typing, a prominent example of a field in which we can apply semantic lifting, has a body of established theoretical and practical work. Session typing was first introduced by Honda [54]. This work also introduced session type inference for the π-calculus. Dezani-Ciancaglini et al [34] brought session types to the imperative world with the language MOOSE. They [33] later expanded upon this work with a notion of compatibility [44].

Let us now compare two projects with similar goals to our implementation as shall be described in Section 4. An implementation of session typing for the Java language appears in Hu et al [56]. In contrast to our system, a domain-specific language approach is employed using the Polyglot [72] front-end. Furthermore, following the approach of Gay et al [44, 45], choices are denoted by labels, whereas our system uses unlabelled, implicit choice. The system also supports higher order session types, essentially the ability for a session to send and receive other sessions, whose types are specified within the enclosing session type.

A further implementation for the Haskell programming language is proposed by Sackman and Eisenbach [79]. As with our approach, session types are integrated into the language's type system, thus no special type checkers or compilers are necessary. Choice is denoted using `offer` and `select` constructs within the session type, similarly to the & and $\oplus$ binary operators of Honda [54]. Higher order session types are supported, as with Hu.

Other means of specifying and verifying protocols for compatibility include finite state automata (including interface automata [31] and choreography [41]), channel contracts [57] and component interfaces [17].

3.7 The GNU Compiler Collection

Due to the open source nature of the GNU Compiler Collection, there has been a great deal of compiler research that has involved modifying its source code in order to provide various extensions to the languages it supports. Indeed, the GCC developers have produced a document providing details of GCC internals [43] in order to provide a starting point for those developers wishing to modify the source code. Examples of extensions that have relevance to this work include compile-time analysis [35], runtime security checks [84] and type system augmentation [12].

3.8 Theorem Provers

There exist a number of *extended static checkers*, static verification tools which use theorem proving techniques in order to verify program properties. The first

of these was implemented for Modula-3 by Detlefs et al [32]. Flanagan et al [39] went on to implement a similar tool for Java.

The tools cited above operate as a verification tool for the programmer, in the case where the program is trusted by the end user. However theorem proving techniques may also be used in order to provide verification capabilities to an end user. This is most useful where the program is not trusted by the user, and the user wishes to verify that it conforms to some desired property, such as correctness or security. Necula [71] describe their work on Proof-Carrying Code, which accomplishes just this, even in cases where the source code is unavailable by the end user. Here the theorem prover is operated by the programmer with full access to the source code. The proof is then encoded with the object code, and is verified by the end-user with reference to the low-level processor instructions present in the object file.

3.9 Ownership Types

Ownership types were introduced by Clarke et al [23] in 1998, in the context of a Java-like language. They give an informal introduction to the concept, followed by a formal type system definition and subject reduction proof. Later work by Boyapati et al [15] adds more features to the ownership type language to make it more resemble Java, such as inheritance and inner classes, to provide the necessary support for efficient iterators, which was not possible under Clarke et al.

Boyapati et al [14] present an interesting reworking of the ownership type concept in the context of deadlock freedom. In their system, lock levels replace ownership contexts, where the lock associated with a particular lock level must be obtained before the objects within that lock level may be accessed. There must also be a partial ordering of lock levels, where the locks must be obtained in the correct order. Boyapati et al's implementation of this system is as a static checker for an extension of the Java programming language. In this way, we can see how ownership types can be applied in a variety of contexts to verify semantic properties.

Andreae et al [4] present a related piece of work that introduced *scoped types* to the Java language. The purpose of this system is to optimise garbage collection in the context of small-scale (embedded) Java systems by assigning scopes to related packages. The scopes are defined in terms of the package layout; similarly to ownership types, the type structure dictates the accessibility of various parts of the program to one another. As well as the active library goal of optimisation, we also gain the verification benefits of ownership types due to the restricted scope.

3.10 Semantic Lifting

We have cited here one particular work that illustrates our concepts of semantic lifting and semantic hinting, but does not fall into a particular category or significant body of work in and of itself. We shall proceed to consider how the concepts we have looked at so far assist us in our goals.

Gil and Lenz [46] describe their system, AraRat, for constructing SQL queries within C++ using a syntax similar to that used in the field of relational algebra. An optional capability of this work is the ability to extract the database schema to be used at compile time for verification purposes. We can consider that this extraction capability allows for the semantic type information to be retrieved from the database and made available to the compiler, thus constituting semantic lifting.

In this chapter we have looked at a number of works in a variety of areas. The overall goal was to attempt to pick out those works which exemplify our concepts of semantic lifting and semantic hinting. To summarise:

- Semantic information enhances the analysis capabilities of active libraries, both within and without the Veldhuizen definition.
- A pluggable type can reflect a semantic property of how values with that type are used. Likewise, semantic lifting can be used to verify this property.
- Abstract interpretation, model checking and theorem provers are three viable techniques for extracting semantic information about a program.
- Meta-object protocols provide an enabling mechanism by which one may extract the information required for semantic lifting. Moreover, a meta-object protocol allows for programs to be manipulated to reflect the semantic properties extracted.
- Session types exist as an expression of the semantic content of a particular dyadic communication, i.e. semantic hinting. They may also be verified in a semantic lifting process.
- By extending a compiler, such as the GNU Compiler Collection, one may gain access to information that can be used in the semantic analysis phase.
- Ownership types create boundaries between objects. These boundaries improve the program verification process. They also necessarily reflect the semantic structure of the program.

Chapter 4

Session Types from Control Flow

This chapter describes our theoretical research relating to session type inference. It has previously been published as a workshop paper at FESCA 2008 [26]. This research is presented in order to motivate the concept of semantic lifting and to provide a basis for the further research work outlined in this report.

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

4.1 Introduction

In order to motivate this case study, let us presently consider a session type consisting of a graph where a communicating process is associated with a single node in a graph. A communication action must be conformant with an outgoing arc at the process's current node, and causes the appropriate arc of the session type to be followed, based on the type of the communication action. Figure 4.1 shows three session type graphs (a), (b) and (c), of which (a) and (b), and (a) and (c) are purportedly compatible with each other (due to the common subgraph). Let us consider two processes A and B with respective session type graphs (a) and (b). Both processes start in state s_1. Firstly process A sends a message of type α and transitions to state s_4. When process B receives this message it transitions to state s_2. Process B then sends a message of type β. However, process A cannot process this message as, according to its session type graph, it may only receive messages of type γ. A similar situation arises with interacting processes A and C of types (a) and (c) where process A first

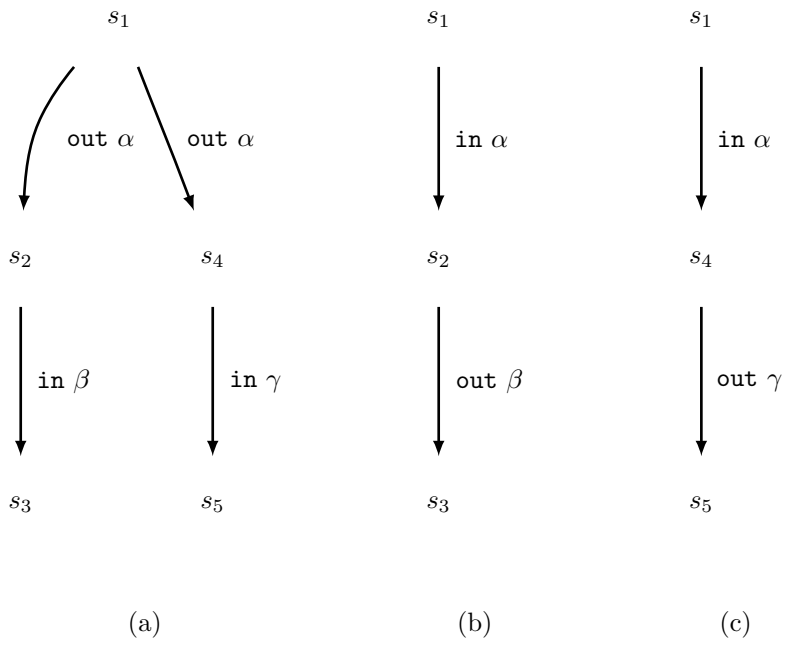

Figure 4.1: Three exemplary session type graphs.

transitions to state s_2 upon receiving the message of type α. We can thus conclude that it is impossible to construct a session type such that processes with that session type may safely communicate with processes with session type (a).

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

We claim the following contributions:

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

4.1.1 Background: implicit choice in session types

As previously discussed, a session type may consist of a choice between a number of other session types. A process may commit to one of these choices either passively or actively, and either implicitly or explicitly. If a process makes the choice actively, then the choice was made by the process based on its choice of communication steps. If the process makes the choice passively, then the choice was made based upon the active choice made by its peer.

Under implicit choice, the process performs a communication action that is consistent with only one of the choices. Note that the process's role in committing to the choice is either active or passive depending on the direction of the communication action. If the process transmits data, its role is active; if it receives data, its role is passive and its choice depends on the type of the data received. [33] is an example of a system which contains a form of implicit choice.

Under explicit choice, the process performs some action other than a communication action that has the effect of selecting a particular choice. The literature includes a number of ways of expressing explicit choice. In [54], choice is represented by the & and $\oplus$ binary operators. A process whose session type is of the form $s_1 \& s_2$ makes a passive choice between s_1 and s_2, whereas a process whose session type is of the form $s_1 \oplus s_2$ makes an active choice via the inl and inr

operators. In [44, 45], each choice is annotated with a *label*. The process making the active choice transmits the label corresponding to its desired choice, and the process making the passive choice chooses the session type corresponding to the label it receives.

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

Compatibility [44] is a relation between session types that indicates whether two programs with specified session types are guaranteed to communicate with each other safely; that is, without any possible protocol incompatibilities at runtime. The compatibility relation has in particular been useful in specifying and verifying *contracts* between two parties: by verifying compatibility before a potential communication takes place it is possible to check that no protocol incompatibilities may possibly occur between the two parties – provided that both parties abide by their session type contracts. It is clear that a necessary condition for a session type to be compatible with another is that it must accept at least the data types which the other may emit. We shall see a more formal definition of this concept later.

The goal of this work is to investigate means for inferring a session type using program analysis techniques given an imperative program consisting of a sequence of communication actions. In some process formalisms, such as the π-calculus as described in [54], there is normally no need for an inference algorithm, as the construction rules for a process implicitly perform typing. Here we adopt a language-neutral approach better suited to the structure of imperative programs, using control flow and expression typing information provided by the host language to derive an appropriate session type. In contrast to many other studies of session type inference [54, 34], our session types use implicit choice. Our rationale for this design decision is that implicit choice provides a closer mapping between the behaviour of the program and its session type. Additionally, it frees the programmer from the burden of providing a tag name for each communication action in an untyped program. We shall explore the consequences of this decision on our type inference technique.

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

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

Figure 4.2: Simple pseudocode server process.

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

Figure 4.3: Simple pseudocode client process.

sary foundation of such compatibility checking being the ability to augment, or simulate the augmentation of, the host language's type system to recognise the session type's subtyping relation.

4.1.2 Example

Consider the server program shown in Figure 4.2, which we wish to interface with the client program shown in Figure 4.3. We verify by inspection that these two programs will interface with each other correctly, and so does our system by means of session type inference and compatibility checking.

Our system can infer the types of both processes. The inferred type for session s in Figure 4.2 is

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

and the inferred type of session s in Figure 4.3 is

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

VT	::=	"int" | "bool" | ...	
TV	::=	$t, t', t''...$	
D	::=	"in" | "out"	
ST	::=	"μ" TV "." ST	(Mu)
	|	"end"	(End)
	|	TV	(TV)
	|	"(" ST "|" ST ")"	(Choice)
	|	"(" ST "." ST ")"	(Seq)
	|	$D\ VT$	(Action)

Figure 4.4: Syntax for a Ninja' session type.

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

4.1.3 Definitions

Our inference system is specified in two distinct ways. Firstly we shall provide a set of inference rules and a methodology for applying them in order to derive a session type. Secondly we shall describe a graph-based implementation technique for the algorithm. The graphs used by this technique are based on finite automata [78] and thus we employ a number of techniques from this field, including the subset construction [78].

Ninja [55] is a specification for a component-based imperative language extension. Our language takes inspiration from this specification, and thus we have named it Ninja'. Ninja' can be considered an implementation of common component models such as architecture description languages as shall be described in Section 4.2. It may extend most imperative languages, however our implementation is for the C++ language and is known as Ninja'-C++. We describe the implementation of Ninja'-C++ and of a type inference tool for it.

Figure 4.4 shows the syntax for session types in the Ninja' language. Note that in informal discussions we use the associativity of "|" and "." to elide parentheses wherever possible. Most of the semantics is clear with reference to Section 4.1, however note the syntax elements (Mu) and (TV). These are standard [76] syntax elements used for recursive type definitions. (Mu) declares a type variable TV of arbitrary name for use, however for convenience we restrict ourselves to the names t, t', t''.... Corresponding (TV) elements are found within the (Mu) element and are equivalent to the whole of the corresponding outer (Mu). In our variant of the syntax, we permit the left-hand side of a sequential composition to be any session type, rather than just an action. This was done for two reasons; firstly to increase expressibility and secondly to allow greater affinity with the type graph used in our implementation (see Section 4.4).

Ninja' is a component based language; components are active and are known as *participants*. Participants communicate with each other over channels of specified session types, which means their session types must be compatible.

We proceed to introduce our notion of compatibility as initially defined by [44] and extended by, among others, [88]. In order to determine compatibility we must first define equivalence, continuation, subtyping and duality for our session type syntax. Equivalence ($\equiv$) is the smallest relation that satisfies the rules given in Figure 4.5. The continuation type of a session type under a given communication action may be derived using the rules given in Figure 4.6.

Many of the equivalence and continuation rules are self explanatory, however we feel it necessary to give a justification of rule (|Dist $\leftarrow$). This will be done in Section 4.3.1 after the necessary background has been described.

Definition 4.1.1. Free names over session types.

$$\begin{array}{rcl}
\mathsf{FN}(\mu t.s) & = & \mathsf{FN}(s) \setminus t \\
\mathsf{FN}(\mathtt{end}) & = & \emptyset \\
\mathsf{FN}(t) & = & \{t\}, t \text{ a type variable} \\
\mathsf{FN}((s_1|s_2)) & = & \mathsf{FN}(s_1) \cup \mathsf{FN}(s_2) \\
\mathsf{FN}((s_1.s_2)) & = & \mathsf{FN}(s_1) \cup \mathsf{FN}(s_2) \\
\mathsf{FN}(a) & = & \emptyset
\end{array}$$

Definition 4.1.2. Input and output domains.

$$\begin{array}{rcl|rcl}
\mathsf{idom}(\mu t.s) & = & \mathsf{idom}(s) & \mathsf{odom}(\mu t.s) & = & \mathsf{odom}(s) \\
\mathsf{idom}(\mathtt{end}) & = & \emptyset & \mathsf{odom}(\mathtt{end}) & = & \emptyset \\
\mathsf{idom}(t) & = & \emptyset, t \text{ a type variable} & \mathsf{odom}(t) & = & \emptyset, t \text{ a type variable} \\
\mathsf{idom}((s_1|s_2)) & = & \mathsf{idom}(s_1) \cup \mathsf{idom}(s_2) & \mathsf{odom}((s_1|s_2)) & = & \mathsf{odom}(s_1) \cup \mathsf{odom}(s_2) \\
\mathsf{idom}((s_1.s_2)) & = & \mathsf{idom}(s_1) & \mathsf{odom}((s_1.s_2)) & = & \mathsf{odom}(s_1) \\
\mathsf{idom}(\mathtt{in}\ v) & = & \{v\} & \mathsf{odom}(\mathtt{in}\ v) & = & \emptyset \\
\mathsf{idom}(\mathtt{out}\ v) & = & \emptyset & \mathsf{odom}(\mathtt{out}\ v) & = & \{v\}
\end{array}$$

Definition 4.1.3. Type simulation [44]. A type simulation is a relation R that satisfies the following property.

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

Definition 4.1.4. Subtyping[1]. $S_1 \leq S_2$ iff there exists a type simulation R such that $(S_1, S_2) \in R$.

Definition 4.1.5. Duality.

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

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

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

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

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

$$\frac{w \notin \mathsf{FN}(S)}{\mu t.S \equiv \mu t'.(S[t'/t])}\ (\mu\text{Ren}) \qquad \frac{S_1 \equiv S_1'}{(S_1.S_2) \equiv (S_1'.S_2)}\ (.\text{Cong} \leftarrow)$$

$$\frac{}{\mu t.S \equiv S[\mu t.S/t]}\ (\mu\text{Exp}) \qquad \frac{S_2 \equiv S_2'}{(S_1.S_2) \equiv (S_1.S_2')}\ (.\text{Cong} \rightarrow)$$

$$\frac{}{(S|S) \equiv S}\ (|\text{Idem}) \qquad \frac{S \equiv S'}{\mu t.S \equiv \mu v.S'}\ (\mu\text{Cong})$$

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

$$\frac{}{((S_1|S_2)|S_3) \equiv (S_1|(S_2|S_3))}\ (|\text{Assoc})$$

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

$$\frac{}{((S_1.S_2).S_3) \equiv (S_1.(S_2.S_3))}\ (.\text{Assoc})$$

Figure 4.5: Rules for equivalence

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

Figure 4.6: Rules for continuation

Definition 4.1.6. Compatibility.

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

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

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

Definition 4.1.7. Safe directionality. A session S is safe-directional iff

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

4.1.4 Type Mutation and Linearity

Throughout this paper, we assume a statically typed language. However, session type theory [54] states that after a session has performed a communication action, its type must automatically mutate to the session's continuation type relative to the action that has taken place. Most statically typed languages do not permit a variable's type to mutate under any circumstances, although some do allow for a variable to be overridden by one with the same name but a more restrictive scope. This seems to be the only practical way to simulate type 'mutation', but the requirement to create a new scope after every communication operation would severely restrict the structure of a program. So we adopt the strategy of introducing a new session variable after each communication action.

After we have used a session variable (i.e. by sending or receiving over it), it becomes invalid. This means that any further use of the variable is an error and would violate our typing system. A variable with such a constraint imposed upon it is known as [94] a *linear variable*, and any program that satisfies this property is said to satisfy the *linearity constraint*. We have developed a prototype tool to check linear usage of session variables [27].

4.1.5 Closing a Session

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

$$c \text{ pdom } a$$

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

Figure 4.7: An example of a Darwin component type (courtesy [67])

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

Figure 4.8: Syntax of session statements. Greyed out syntax is not present in the input data.

i.e. c postdominates [2] a. Intuitively this means that all sessions that are scheduled to close (by a communication operation resulting in a session of type **end**) are guaranteed to close by the control flow of the program, provided the program is not interrupted, e.g. by the operating system.

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

4.2 Related Work

Ninja' provides a component model similar to that of the Unified Modeling Language [73] or architecture description languages such as Darwin [67]. While the UML component model largely deals in the abstract, permitting any form of communication such as a streaming model, shared memory model or procedure calls, Ninja''s model, similar to Darwin's, restricts communication to a streaming model using the provided communication channels. Darwin's communication channels have a simple notion of typing as shown in the example component type of Figure 4.7, however the session typed nature of Ninja''s channels affords a greater deal of flexibility.

4.3 Derivation and Canonicity

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

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

The syntax for program statements that operate on sessions is, however, defined by the syntax given in Figure 4.8. Our algorithm supports an unbounded number of concurrent sessions.

Definition 4.3.1. Choice composition. The choice composition operator $\Big|$ is defined nondeterministically as follows.

$$\Big|S \quad = \quad \begin{cases} (s\Big|(S \setminus \{s\})), & |S| \geq 2 \wedge s \in S \\ s, & S = \{s\} \end{cases}$$

Definition 4.3.2. Canonicity.

1. $(S_1|S_2)$ is canonical if sessions S_1 and S_2 are canonical, $\mathsf{idom}(S_1) \cap \mathsf{idom}(S_2) = \mathsf{odom}(S_1) \cap \mathsf{odom}(S_2) = \emptyset$, $S_1 \not\equiv \texttt{end}$ and $S_2 \not\equiv \texttt{end}$.
2. $(a.S)$ is canonical where a is an action as defined in Figure 4.4 and S is canonical.
3. $\mu t.S$ is canonical if S is canonical and $t \in \mathsf{FV}(S)$.
4. $\texttt{end}$ is canonical.
5. t is canonical, t a type variable as defined in Figure 4.4.

We begin by rewriting all statements of form $[\![sv\texttt{.send}(x)]\!]$ to $[\![sv\ \texttt{:=}\ sv\texttt{.send}(x)]\!]$; and all statements of form $[\![sv\texttt{.receive}(x)]\!]$ to $[\![sv\ \texttt{:=}\ sv\texttt{.receive}(x)]\!]$. We proceed to convert session statements to single static use [66] form. The rules given in Figure 4.9 are then applied to assign a type to each session variable by solving for $\Delta = \emptyset$ where Γ contains language-specific typing information for the current context. Each well-formed type must have a *canonical* form as described in Definition 4.3.2, which is equivalent to the original derived type according to the equivalence rules given in Figure 4.5. If any type is not well-formed, i.e. it does not have an equivalent canonical form, the inference algorithm fails. After the canonical form for each session type is derived, we eliminate λ statements by first globally replacing any session variable appearing on the left hand side of a λ statement with the session variable named on the right hand side, then removing the λ statements themselves. Note that session variables retain the type assigned to them before λ statements were eliminated.

4.3.1 Justification

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

Canonicity rule 1 ensures that no two alternatives in a choice construct may present the same choices. This rule ensures the deferment of such choices to the

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

$$\frac{\begin{array}{c}[\![sv' := sv.\texttt{send}(x)]\!] \in \mathsf{Stmts} \quad \Gamma \vdash x : v \quad t \text{ fresh} \\ \Gamma, \Delta[sv \mapsto t] \vdash sv' : s' \quad s \notin \text{dom } \Delta\end{array}}{\Gamma, \Delta \vdash sv : \mu t.(\texttt{out } v.s')} \text{ (Send)}$$

$$\frac{\begin{array}{c}[\![sv' := sv.\texttt{receive}(x)]\!] \in \mathsf{Stmts} \quad \Gamma \vdash x : v \quad t \text{ fresh} \\ \Gamma, \Delta[sv \mapsto t] \vdash sv' : s' \quad s \notin \text{dom } \Delta\end{array}}{\Gamma, \Delta \vdash sv : \mu t.(\texttt{in } v.s')} \text{ (Recv)}$$

$$\frac{\begin{array}{c}[\![\lambda(sv_1, sv_2, \ldots, sv_n) := sv]\!] \in \mathsf{Stmts} \quad t \text{ fresh} \\ \forall 1 \leq i \leq n : \Gamma, \Delta[sv \mapsto t] \vdash sv : s_i\end{array}}{\Gamma, \Delta \vdash sv : \mu t.(\,|\{s_i : 1 \leq i \leq n\})} \text{ (Lambda)}$$

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

Figure 4.9: Type inference rules

last possible moment. This reflects the restrictions imposed on the communicating process, namely that a process may only choose which branch it takes on the basis of the type of the variable it sends or receives, and not any other information. In the process of applying equivalence rules to a session type in order for it to conform with canonicity rule 1, equivalence rule ($|\text{Dist} \leftarrow$) will be most frequently employed. This rule prevents the situation shown in Section 4.1 where two distinct branches of a session type are initially distinguished by the types of their inputs. There is no need to impose such a rule on branches which are initially distinguished by the types of their outputs, as a communicating process may simply accept both value types at this point.

4.4 Algorithm

This section supplies a concrete description of our type inference algorithm suitable for implementation. Our algorithm is implemented in three stages. For the purpose of illustration we shall use a simplified version of Ninja'-C++ called $\mathcal{L}_{\mathsf{N}}$ whose syntax contains only `if`, `while` and session communication statements with the symbol $*$ substituted for boolean expressions and expression types substituted for all other expressions and whose control flow is defined in the obvious way. It is possible to translate a Ninja'-C++ program written in C++ into $\mathcal{L}_{\mathsf{N}}$ by converting `for` loops into `while` loops in the usual way, removing all variable declarations, removing all statements without a counterpart in $\mathcal{L}_{\mathsf{N}}$ and

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

Figure 4.10: Simple $\mathcal{L}_{\mathsf{N}}$ program.

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

Figure 4.11: Simple $\mathcal{L}_{\mathsf{N}}$ program after SSU applied.

replacing all primitive values with their types. Note that $\mathcal{L}_{\mathsf{N}}$ does not include invocations because we are not inferring the type of the channel; it may have any type less specific than the participant's dual and more specific than the invoker's, and compatibility between participants and invokers is achieved by upcasting the return value from the `invoke` method into the appropriate type. Our language supports an unbounded number of concurrent sessions.

4.4.1 Stage 1: Static Single Use

The first step is to ensure that no session variable is reused more than is necessary. This is different from the linearity constraint mentioned in Section 4.1.4; what we would like to do here is to detect legitimate, linear programs that reuse session variables instead of using a fresh variable wherever possible, meaning that our inference algorithm would generate too general a session type. In the most extreme case, only one session variable is used throughout an entire procedure (note that this is the starting point of our derivation algorithm). Thus the program's communication statements are first converted to SSU [66] form. Figure 4.10 shows a program in $\mathcal{L}_{\mathsf{N}}$ with liberal reuse of session types, and Figure 4.11 shows the same program after SSU has been applied to it.

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

Figure 4.12: Graph building function g.

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

Figure 4.13: Unification mapper and helper function ϵ.

4.4.2 Stage 2: Graph Building

After obtaining the SSU form of the program, we then build a graph of the session transitions contained within the program using its communication statements. The function g that builds this graph is shown in Figure 4.12, assisted by the *unification mapper* f_G shown in Figure 4.13. The goal of this function is twofold:

- to extract all communication actions and collect them into a graph with arcs between source and target session variables;
- for variable assignments, ensure that the source and target sessions receive the same type (this is the purpose of the δ function built by f_G). The helper function ϵ (Figure 4.13) assists in this by providing a means for a given set of variables to receive the same type.

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

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

Figure 4.14: Session type building function τ.

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

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

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

4.4.3 Stage 3: Graph Simplification and Translation

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

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

Definition 4.4.1. Node equivalence within a graph.

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

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

Definition 4.4.2. Applying a substitution function. To apply a substitution function δ, we replace the current graph G with the result of unification mapper $f_G(G, \delta)$, where f_G is defined in Figure 4.13.

We may unify nodes provided that they are node equivalent, according to Definition 4.4.1. This allows us to simplify graphs with multiple convergent arcs with the same label leading to a single node. For each pair of nodes n_1 and n_2 in our graph G such that $n_1 \equiv_G n_2$, we apply the substitution function $\epsilon(n_1 \cup n_2)$.

A second case we must deal with is divergence. Should a graph have many divergent arcs with the same label leading to nodes $n_1, n_2 ... n_n$, we must replace these nodes and any dependent subgraph with a single node n and associated

subgraph such that $\forall i \in \{1...n\}, \tau(n) \leq \tau(n_i)$, where τ is the session type building function defined in Figure 4.14. The initial processing may be achieved by treating our session graph as a NFA, converting it into a DFA using the subset construction and rejecting any graph that does not satisfy this property. This transformation is sound as it has been proven [78] that each NFA has an equivalent DFA (accepting the same language or, in our case, sequence of communication actions) which translates directly to trace soundness.

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

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

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

Definition 4.4.3. Superficial subtyping. A type graph $H = (N_H, E_H, A_H)$ is a superficial subtype of a type graph $G = (N_G, E_G, A_G)$ iff:

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

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

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

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

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

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

We may now extract the session types from our graph by employing the τ function shown in Figure 4.14 and using equivalence rules, in particular μExp and congruence, in order to eliminate unnecessary μ operators.

4.5 Example

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

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

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

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

We proceed to stage 1, converting to SSU form:

```
while (*) {
  s2 := s1.receive(Req2) ;
  λ(s3, s4) := s2 ;
  if (*) {
    s5 := s3.send(int) ;
    s6 := s5.receive(int) ;
    s1 := s6.send(char)
  } else {
    s7 := s4.send(int) ;
    s8 := s7.receive(int) ;
    s1 := s8.send(long)
  }
```

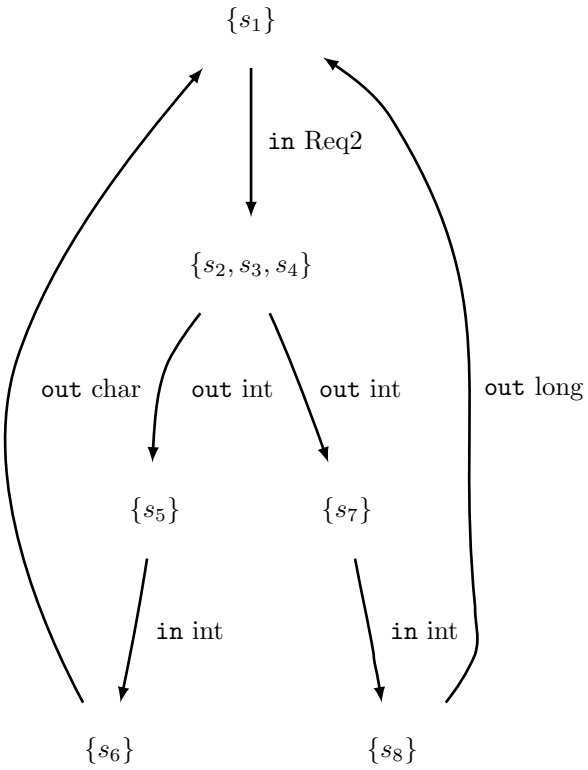

Figure 4.15: Result of graph building function f_G applied to Example 1.

}

Applying the graph building function f_G we obtain the graph shown in Figure 4.15. This graph has no accepting states so the definite termination property vacuously holds.

The graph has no recursively equal nodes for us to unify, so we proceed to DFA building using the subset construction, giving us the graph shown in Figure 4.16. In this graph, each node is a disjoint subset of the set of sessions, so our substitution function has no effect. Applying the τ function to our graph to produce a Ninja' session type, we deduce the following overall type assignment for s_1:

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

4.6 Sessions in C++

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

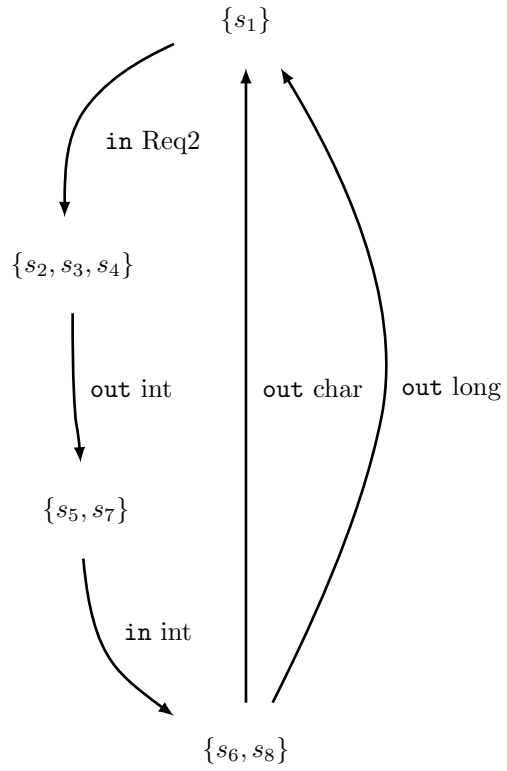

Figure 4.16: Result of subset construction applied to Figure 4.15.

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

Figure 4.17: A recursive session type representing an infinite stream of `int` inputs

language extensions.

4.6.1 Sessions and Channels

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

We distinguish between Ninja' *actions* and *sessions*. An action is a primitive communication step, such as **in int** (of the form $D\,V\,T$ from Figure 4.4), whereas a session is a fully specified session which may include sequential composition, choice etc. Actions in Ninja'-C++ shall take the form **in**<T> or **out**<T>, where T is the primitive data type to be sent or received across the channel.

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

Frequently when designing session types, we must be able to create recursive types. This will commonly occur when we would like to represent a loop in our session typed code (for example a request-response loop, or a computation which may produce an arbitrary number of responses). The most obvious way of creating a recursive type in C++ (that is, defining a type in terms of itself in a **typedef** statement) will not work, because the language prevents such a definition. However an incompletely-defined type may be referred to in a template instantiation. This allows us to specify a three-stage protocol that may be used to define a recursive type. Firstly, an incomplete struct s is defined. Secondly, the recursive session type r is defined using a **typedef**. Wherever a recursive reference is required, the special instantiation **call**<s> is used. Thirdly, s is fully defined, with an internal typedef t that is defined to be r. An example of such a definition is shown in Figure 4.17.

What we have done in the previous paragraph is establish an *isorecursive*

type system [76]. As opposed to the equirecursive type system of Ninja', where a recursive type and references to the recursive type are equivalent via the (μExp) rule given in Figure 4.5, in our isorecursive type system we have established an isomorphism between the 'rolled' reference type **call**<s> and the 'unrolled' type r. The 'unroll' operation is carried out automatically during the computation of the continuation type of a particular session type if it is found to be of the form **call**<s>. In this case there is no inverse mapping from unrolled types to rolled types; we do not require one here, but it would be trivial to define one in order to make this a 'true' isorecursive type system.

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

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

4.6.2 Participants

Each participant comprises:

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

and provides the following functionality:

- a constructor which is provided with a sequential list of channels in the order provided by its definition;
- a `spawn` method which spawns the participant.

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

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

```
struct part_base {

  channel<s1> *ch1;

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

};

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

Figure 4.18: An example of a skeleton participant

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

An example of a skeleton participant is shown in Figure 4.18.

4.6.3 A Note on Session Variable Types

As previously mentioned, each session variable must be fully specified with its session type. It is unfortunate that the C++ language does not provide us with the facility of automatically deducing the session variable's type, even though it has all the information available to do so. The most recent draft of the C++ standard [58] provides for an `auto` specifier for variable declarations (section 7.1.5.4) which deduces the type of a variable from the type of its initialiser. This would be ideal for our purposes here, but since the document is still in draft, no compiler implements this feature yet, and we have to make do with what we have.

4.7 Implementation

Our prototype implementation of this algorithm covers stages 2 and 3 of the algorithm described in Section 4.4, with two crucial differences:

- As Ninja'-C++ does not currently take into account session subtyping as described in Definition 4.1.4, an invoker's communications must produce the exact same session type via our algorithm as the dual of the corresponding communication procedure for them to be compatible.
- It only performs a simplified version of the subset construction, and does not check the superficial subtyping property for minimised graphs.

It is a C++ program transformation using the ROSE [81] source-to-source translator framework. The transformation takes an untyped Ninja'-C++ program as input, and generates a compilable typed program as output.

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

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

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

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

4.8 Conclusion and Future Work

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

In the context of this research work, the Ninja'-C++ implementation can be considered an active library outside of the Veldhuizen definition, This is because its primary purpose is to verify that a certain chain of communication actions fulfills the requirements of a particular session type; thus it acts in a verification role, rather than an optimisation role. Furthermore, we can consider the type inference algorithm and tool implementation to be carrying out a form of semantic lifting, as the semantic properties of the communication flow between the two peers is extracted.

Ninja'-C++ does not currently decide compatibility according to Definition 4.1.6; instead, two session types are deemed to be compatible only if they are the exact dual of each other. Clearly, this does not afford us much flexibility. The reason for this is that any such compatibility check, being a compile-time mechanism, must take place within the language's facilities for compile-time computation. For C++, this means the template system. However, the C++ template system, despite being Turing complete [89], has insufficient expressibility for a maintainable implementation of the compatibility relation to be feasible. In order to add a dynamic layer of expressibility to the language, a compile-time extension framework can be implemented providing computed template instantiations in a functional, or semi-functional, language such as ML or Haskell. In this instance, the extension framework can be used to build a template representing a binary relation of session subtyping as described in Definition 4.1.4. We can then use custom type conversion operators and the Substitution Failure Is Not An Error (SFINAE) principle to facilitate substitutability and thus, by the construction of Ninja'-C++, compatibility. Chapter 5 of this report describes preliminary work in this area which will eventually allow us to implement the ideas described in this paragraph.

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

Chapter 5

A C++ Meta-Object Protocol using Haskell

C++ metaprogramming is sometimes criticised for being rather obtuse and unmaintainable. Much of this criticism stems from the syntax used to express these computations, which appears unintuitive to those not intimately familiar with the C++ standard. Furthermore, there exist certain computations which remain nearly impossible to express using metaprogramming, for which an easier alternative is desired. From examining other existing metaprogramming systems, such as LISP and Template Haskell, it is noted that these languages expose an abstract syntax tree representation of the language to computation in a full-featured standard programming language. In both of these cases, we note that the language in question is the same as the target language. This is ideal for a functional language, because symbolic manipulation, as commonly found in functional languages, is required in order to produce the desired result.

The current situation with C++ is that template metaprogramming generally allows for computations to be performed over types and no representation of the rest of language is available for direct manipulation. We may consider C++ template metaprogramming to be a (simplified) functional language in the domain of C++ types. There is therefore scope for expansion in the language – namely, we wish to provide a framework for an alternative, more powerful programming language to play the role of computing a type in the same way that template metaprogramming currently does. This framework shall be provided as an extension to the C++ front-end of the GNU Compiler Collection, or GCC.

One may consider our language extension to be a form of Meta-Object Protocol or MOP. We contrast this work from existing MOPs for C++, such as MPC++ [59] and OpenC++ [21], as our language extension constitutes an enhancement to the existing concept of MOPs as provided in those language extensions. The key distinction is that the MOP may play an active role in the operation of the existing meta-level architecture. This can be compared to the

concept of active libraries, which play an active role in a program's compilation. Where the meta-level architecture is limited, which is the case with C++ templates, the enhanced MOP becomes particularly valuable, and can substantially assist in the implementation of active libraries, if a sufficiently powerful language is chosen as our meta-language.

We aimed to choose a language for this purpose which was as similar to the C++ template metaprogramming "language" as possible, in order that its integration into the language may be as unobtrusive as possible. In particular, we considered the following attributes:

Symbolic Manipulation The C++ template programming language supports pattern matching for template arguments. It also supports a shorthand method of building complex structures of template instantiations, namely the standard textual representation of that structure.

Purity The C++ template metaprogramming language is pure. This means that providing the same arguments to a template will always result in the same instantiation.

Instantiation A C++ template instantiation may instantiate other templates and use the results of these instantiations to carry out further computation.

By considering these attributes and various other considerations we decided upon the following key criteria:

Symbolic Manipulation We would like to preserve the pattern matching and type structure building features in our target language in order to keep manipulation rules easy to write.

Purity In order to remain consistent with the unaugmented language, we would prefer to retain this purity property in our language.

Foreign Function Interface GCC is written in C. The language must, therefore, have the capability to be called from C. Conversely, in order for the target language to interrogate GCC's internal data structures and instantiate templates itself it must be able to call C code.

The most prominent "pure" language available today is the functional language, Haskell. Haskell boasts powerful pattern matching and data structure building capabilities. It also sports the concept of *monads* for performing "impure" computations, including calling C functions via its foreign function interface. The decision was thus made to use Haskell for this purpose.

5.1 Design

The current design of the language extension is that the user provides a *transformation function* written in Haskell, which takes a Haskell representation of

```
data CTTree = CTString String
            | CTInt Int
            | CTPtr CTTree
            | CTRef CTTree
            | CTNamed String [CTTree]
            | CTTemplate String
              deriving (Show, Eq)
```

Figure 5.1: Type specification for `CTTree`.

a C++ type (a `CTTree`) and returns another `CTTree`. In order to use the language extension, a user annotates a `typedef` statement with a custom attribute containing enough information to allow the C++ compiler to locate the implementation of the transformation function.

The type specification for `CTTree` is given in Figure 5.1. The `CTString`, `CTInt` and `CTTemplate` choices represent template parameters only, whereas `CTNamed`, `CTPtr` and `CTRef` may also represent types. Note that it is a current deficiency in the design that a template parameter has the same type as a type and thus invalid types could potentially be constructed (this results in a GCC internal error); we aim to correct this deficiency soon via the use of additional type specifications. Note in particular the `CTString` choice; this represents a small extension to GCC's C++ front-end to allow it to accept string constants as template parameters. Normally such parameters are disallowed because there is no way for the native C++ template metaprogramming mechanism to manipulate them. However, in the context of an extended metaprogramming language, string constants become very useful, as we shall see in our example.

Normally, a C++ `typedef` statement takes two parameters: the *source type* and a *target name*. A standard C++ compiler, when encountering this statement, will insert into the symbol table (taking into account the scope in which the `typedef` statement appears) a new symbol with the same name as the target name; a type symbol equivalent to the source type. Under our type system extension, if the `typedef` is annotated with the above mentioned attribute, the source type will be converted to its Haskell representation and passed to the transformation function. The result of the transformation function will be converted to a C++ type, and it is this type which will receive the alias of the target name in the symbol table.

5.2 Implementation

We have a preliminary implementation of our Haskell extension concept using the development (1st February 2008) version of GCC as a baseline. In our implementation, a number of internal GCC functions are exposed to Haskell via a GCC support module written in Haskell. The role of the support module is to provide a gateway between the functionally pure world of Haskell and the

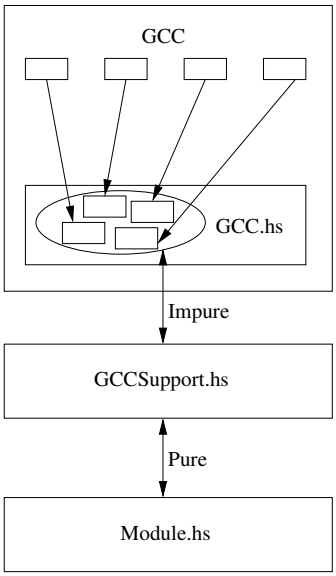

Figure 5.2: Architecture of GCC extension

impure world of GCC's internals. This is achieved by a function, known as `pureAdaptor`, which converts a pure transformation function to an impure one of the form expected by our GCC extension. This function uses GCC internal functions via FFI to convert C++ types to their Haskell representations and back, and to instantiate other templates. The architecture of the implementation is illustrated in Figure 5.2.

The Haskell compiler used in our implementation is GHC, although since the FFI is implementation independent it could conceivably be ported to another Haskell compiler. In order to prepare a transformation function for use by our language extension, it must first be placed in a Haskell source file and compiled using GHC. The custom attribute contained in the relevant `typedef` thus contains the following information in two text strings:

1. The path to the Haskell object file containing the compiled version of the transformation function;

2. The (symbol) name of the transformation function within the object file.

5.3 Example

The example revealed in this section shows how we can create a C++ template which would be impossible to implement using the standard C++ template mechanism. The goal is to parse textual representations of session types into

the hierarchy of template instantiations we have defined for this purpose, as described in Section 4.6.1 of this report. As an example, the type

```
parse<"?int.!int.end">::t
```

will be an alias for

seq<in<int>, seq<out<int>, end>>

In order to pass the string parameter to the Haskell code, we must first "wrap" it in a template. This is because the string constant is a value, and the template is necessary to convert it into a type. The resultant template declaration is shown below.

```
template <const char *ST> parse {

  typedef __attribute__((computed_type_info(
                  "GCCSessionType.o", "parseSessionType"
          ))) string_wrap<ST> t;

};
```

When this template is instantiated, the pure transformation function is evaluated via the support module. Its parameter is a type structure of the form

```
CTNamed "string_wrap" [CTString s]
```

where s is the desired string value (e.g. "?int.!int.end" in our example). The string is then lexed and parsed; for the parsing we use an LALR(1) parser generated by the Happy [68] parser generator. The parser produces a value of type `SessionType` (for generality) which is then converted to type `CTTree` and returned. For our example, the resultant value of type `CTTree` would be

```
CTNamed "seq" [CTNamed "in" [CTNamed "int" []],
    CTNamed "seq" [CTNamed "out" [CTNamed "int" []],
    CTNamed "end" []]]
```

5.4 Conclusion and Future Work

We have presented a mechanism for C++ template metaprogramming using the Haskell programming language, with a workable type representation and usage methodology. This project is in its preliminary stages; there is clearly much work to do, in terms of expanding the type representation, developing interesting case studies and additional language features.

One planned feature is to expand the scope of the extension so that it may be used to define entire template instantiations including constant declarations and inner classes. Not only would this allow for a more usable interface in some cases, but it would also allow us to define more complicated type structures, such as the structures required for the three-stage declaration protocol for isorecursive session types described in section 4.6.1 of this report.

Chapter 6

Plans

In this report the base techniques for my research work, namely active libraries and semantic lifting, have been considered. We have also presented work relating to Ninja'-C++, our C++ implementation of session types, which can be considered an active library outside of the Veldhuizen definition with semantic lifting capabilities. We have also presented preliminary work relating to an enhanced meta-object protocol for C++. My research plans include developing these implementation projects with particular prominence given to active library and semantic lifting techniques. The sections below shall outline exactly how this shall take place.

As shall be outlined in this chapter, my research plans include practical implementation work primarily, as the main focus for my research. However, theoretical aspects shall be touched upon where necessary.

6.1 The Ninja'-C++ Library

In Section 4.8, we have discussed a number of possible improvements to the Ninja'-C++ library, in particular in the area of type inference. As part of this research, I plan to carry out these improvements. As discussed in that section, other work (namely our meta-object protocol for C++) is required for the purpose of implementation. For this reason, the priority shall be on creating an implementation of this extension that can be used for this purpose.

Other planned work, irrelevant to type inference but relevant from the point of view of active libraries and semantic lifting, relates to performance optimisations in the Ninja'-C++ library. In the current implementation of the library, there is scope for greater efficiency in terms of the mechanism by which data is communicated between participants. As discussed in Section 4.6, there exists an implementation of the `send` and `receive` primitives for sending and receiving data. Currently, the implementation performs one physical communication operation (namely a system call) per logical communication operation (namely a call to the `send` or `receive` primitives). From a performance point of view,

this is suboptimal, as under most communication models, the most optimal approach is to perform as few communication operations as possible. Furthermore, where the communication between the peers is characterised by iterations of "back and forth" communication, where there exist no dependencies between each iteration, an unneccessary delay is introduced based on the communication delay. An opportunity arises to optimise these communications, following the work of Yeung and Kelly [99], by amalgamating as many outgoing logical communication operations as possible into a single physical communication operation.

6.2 A C++ Meta-Object Protocol using Haskell

In Section 5.4, we have discussed our planned improvements to our C++ meta-object protocol. As part of this research, I plan to carry out these improvements. As stated in that section, I also plan to develop case studies for the use of this tool. I shall now elaborate on one such case study.

In the context of Remote Method Invocation (RMI), and similar systems such as the Common Object Request Broker Architecture (CORBA) [92], object-oriented programming is placed within a distributed setting. Objects may reside on a remote machine under the aegis of a specialised object server, known under CORBA as the *Object Request Broker*, which provides the necessary mechanism for methods to be invoked remotely. From the client's point of view, there must be a mechanism that allows remote objects to be substituted for local objects, so that the use of remote objects is transparent to the user (to the extent permitted by the particular programming language). In C++, this is usually achieved by the generation of *stubs*. A stub is a fragment of code that carries out the method-specific preparations for the method to be executed remotely.

Stubs are usually created by a specialised program, specific to the implementation in use. However, where a meta-object protocol exists, the stub generator will be unable to "see" any classes generated by the MOP. Furthermore, in the context of C++, stub generators may have difficulty instantiating templates correctly, and thus we may have trouble exporting template instantiations via our mechanism. We thus propose a stub generator that is built using our meta-object protocol. Using the stub generator may be as easy as instantiating a template with the name of the class for which a stub is required as a parameter. The task of this stub generator shall be to enumerate the methods of a particular class as they are visible to the compiler, and generate the necessary stub code. This stub generator would be able to handle any class visible to it, including template instantiations and classes generated dynamically using the meta-object protocol, as it would have access to the definitive compiler interpretation of those classes. A similar mechanism can be used to generate the *skeletons* which are used on the server side to delegate incoming method calls to the appropriate object.

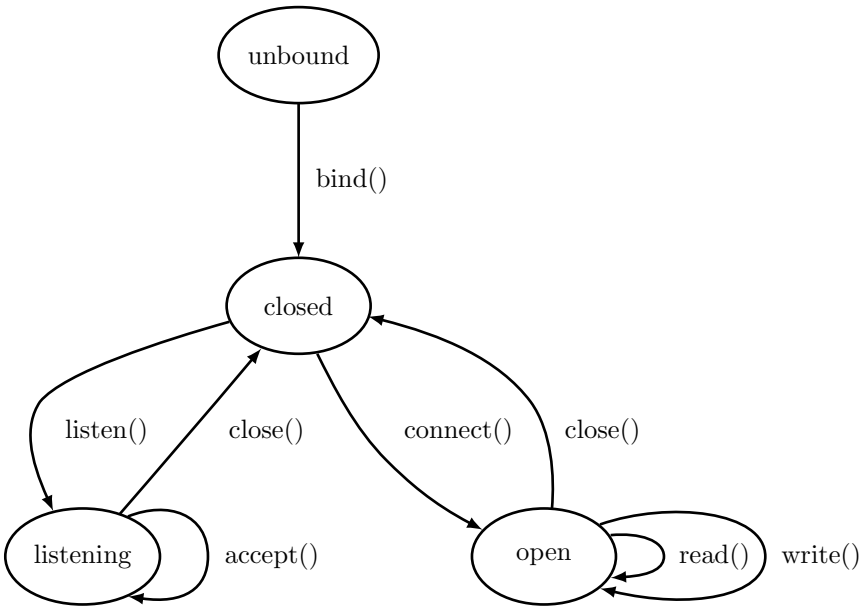

Figure 6.1: Lifecycle of a TCP socket object.

6.3 Session Types, Lifecycle Classes, Type Mutation and Linearity

A planned piece of research concerns an application of the concept of semantic lifting, using some of the concepts explored in our session type work. One may consider a session type to be used to model not only communication protocols but also particular interfaces or objects. Effectively the session type may be used to characterise the permitted use of the object over its lifecycle.

In our system, a session type variable represents the session at a particular communication stage. Furthermore the variable is rendered invalid after a communication action has occurred over it (see Section 4.1.4). By analogy, we can propose a set of classes (which we shall call *lifecycle classes*) corresponding to a particular class (the implementation class), where an instance of a lifecycle class represents an instance of the implementation class at a particular stage of its lifecycle. The lifecycle class instance acts as a handle for the implementation class instance, in that the lifecycle class instances are used directly, which in turn manipulate the implementation class. After the lifecycle class instance is used to manipulate the implementation class instance such that its state changes, the lifecycle class instance is, as with the session type variable, rendered invalid.

One particular example of a situation in which this technique would be useful is a class representing a TCP socket, in which the lifecycle resembles that shown in Figure 6.1. Error handling is omitted for simplicity at this stage.

The question of how to generate these lifecycle classes arises. In order to provide an intuitive interface, it is desirable to use a graph representation as a starting point. The graph would be of the form shown in Figure 6.1 and would include nodes describing which states exist and which methods are available within each state, and edges describing which transitions are possible between states and which methods cause these transitions to occur. Using the information contained within the graph, we proceed to generate a lifecycle class implementation for each node in the graph. There are then two alternatives we can consider: the first is to use a code generator to generate the appropriate implementation textually, the second is to use an MOP, such as the MOP being implemented as part of this research, to generate the appropriate implementation at compile time.

We may also wish to generate this graph ourselves from information contained within the implementation class. This is where our concepts of semantic lifting and semantic hinting enter the picture. From a semantic lifting point of view, from analysing the control and data flow of the implementation class, we can determine which "guards" exist for a procedure and in which circumstances the guards' truth value changes. Or, from a semantic hinting point of view, we can also extract the same information from any pre- and postcondition annotations contained within the program. The exact details of the method of producing the graph from the guards, or preconditions and postconditions shall be a topic for further research.

We shall also need a mechanism for enforcing the linearity constraint upon the lifecycle classes where necessary. In previous work [27], we have described an implementation of a linearity verification tool for the specific purpose of verifying correct use of session variables. In order that it may be put to use in this context, we must develop a general linearity checker which can handle source code annotations indicating which classes are linear, and which methods cause objects of that class to be deemed invalid. The enhanced linearity checker would constitute a tool which uses semantic hinting information.

6.4 Timeline

The following constitutes my plan for the remainder of the degree. As an ongoing project, the research work into developing the Ninja'-C++ library as a whole, including the linearity constraint, shall be considered for publication at a relevant conference or workshop.

September – October 2008 Add the features to the implementation of the inference algorithm for Ninja'-C++ that have relevance to type inference, as discussed in Section 4.8. This includes the more accurate subtyping relation, and possibly delegation. Before this is done however, the C++ MOP shall be developed to have sufficient capabilities to allow these features to be supported.

November – December 2008 Develop our C++ meta-object protocol to the

extent discussed in Section 5.4; the implementation thus derived shall support our later research work as set out below.

January – February 2009 Develop the communication algorithm optimisation discussed in Section 6.1, including considering the best implementation approach – either our C++ meta-object protocol, or a program analysis and transformation framework such as ROSE [81].

March – April 2009 Develop the implementation for lifecycle classes as discussed in Section 6.3, using our C++ meta-object protocol.

May – July 2009 Consider further case studies for our C++ meta-object protocol, as well as our stub generator discussed in Section 6.2, including implementation work.

August – October 2009 Investigate the consequences of augmenting other programming languages with similar or advanced metaprogramming capabilities. The exact details of such augmentation will need to be developed in a sensible manner in order to take account of the features provided by the language, and will reflect the results of exploratory research into the shortcomings of the facilities provided by the languages.

November 2009 – Writeup

Bibliography

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, I. N. I. Adams, D. P. Friedman, E. Kohlbecker, J. G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language Scheme. *SIGPLAN Lisp Pointers*, IV(3):1–55, 1991.

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

[3] P. Alpatov, G. Baker, H. C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. PLAPACK: Parallel linear algebra libraries design overview. 1997.

[4] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time Java memory management. *Real-Time Syst.*, 37(1):1–44, 2007.

[5] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74, 2006.

[6] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, pages 203–213. ACM Press, 2001.

[7] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical report.

[8] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[9] T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264, London, UK, 2001. Springer-Verlag.

[10] R. Barbuti, C. Bernardeschi, and N. D. Francesco. Checking security of Java bytecode by abstract interpretation. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 229–236, New York, NY, USA, 2002. ACM.

[11] M. Barnett, K. Rustan, M. Leino, and W. Schulte. The Spec♯ programming system: An overview. pages 49–69. Springer, 2004.

[12] G. Baumgartner and V. F. Russo. Signatures: a language extension for improving type abstraction and subtype polymorphism in C++. *Softw. Pract. Exper.*, 25(8):863–889, 1995.

[13] K. Böllert. On weaving aspects. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 301–302, London, UK, 1999. Springer-Verlag.

[14] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Nov. 2002.

[15] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, 2003.

[16] G. Bracha. Pluggable type systems. October 2004.

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

[18] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1997.

[19] N. Charlton. Program verification with interacting analysis plugins. *Form. Asp. Comput.*, 19(3):375–399, 2007.

[20] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA, 2002. ACM.

[21] S. Chiba. OpenC++ 2.5 reference manual, 1997.

[22] S. Chiba. Javassist — a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

[23] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, Oct. 18–22 1998. ACM Press.

[24] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[25] E. M. Clarke and B.-H. Schlingloff. Model checking. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1635–1790. Elsevier and MIT Press, 2001.

[26] P. Collingbourne and P. Kelly. Inference of session types from control flow. In *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2008)*, 2008.

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

[28] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[29] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[30] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

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

[32] D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. *Compaq Systems Research Center*, 1998.

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

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

[35] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 1–1, Berkeley, CA, USA, 2000. USENIX Association.

[36] M. D. Ernst. Annotations on Java types: JSR 308 working document. http://pag.csail.mit.edu/jsr308/, November 12, 2007.

[37] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. *SIGPLAN Not.*, 38(11):302–312, 2003.

[38] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2-3):163–189, 1999.

[39] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *SIGPLAN Not.*, 37(5):234–245, 2002.

[40] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 327–335, New York, NY, USA, 1989. ACM.

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

[42] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. *SIGPLAN Not.*, 34(5):192–203, 1999.

[43] Free Software Foundation. *GNU Compiler Collection (GCC) Internals*, 2008.

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

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

[46] J. Y. Gil and K. Lenz. Simple and safe SQL queries with C++ templates. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 13–24, New York, NY, USA, 2007. ACM.

[47] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.

[48] S. Z. Guyer and C. Lin. Broadway: a compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, 2005.

[49] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 448–472, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[50] G. J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice and Experience*, 18:137–161, 1988.

[51] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[52] G. J. Holzmann. Logic verification of ansi-c code with spin. In *Verlag / LNCS 1885*, pages 131–147. Springer, 2000.

[53] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

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

[55] K. Honda, N. Yoshida, and K. Osmond. Outline specification of Ninja, a typed low-level language for chip-level multiprocessing. Unpublished draft.

[56] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In J. Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

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

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

[59] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and implementation of metalevel architecture in C++ – MPC++ approach. In *Reflection '96*, pages 153–166, 1996.

[60] N. D. Jones and S. S. Muchnick. Binding time optimization in programming languages: Some thoughts toward the design of an ideal language. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 77–94, New York, NY, USA, 1976. ACM.

[61] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[62] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[63] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 77–84, New York, NY, USA, 1987. ACM.

[64] S. Krishnamurthi. Educational pearl: Automata via macros. *J. Funct. Program.*, 16(3):253–267, 2006.

[65] A.-F. LeMeur, C. Consel, and B. Escrig. Guaranteed configurability of components via specialization modules. Research Report 1256-01, LaBRI, Bordeaux, France, Mar. 2001.

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

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

[68] S. Marlow and A. Gill. Happy User Guide. `http://www.haskell.org/happy/doc/html/`.

[69] M. D. McIlroy. Macro instruction extensions of compiler languages. *Commun. ACM*, 3(4):214–220, 1960.

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

[71] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[72] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

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

[74] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 22–24, 2008.

[75] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

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

[77] A. Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, 0:46–57, 1977.

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

[79] M. Sackman and S. Eisenbach. Session types in Haskell: Updating message passing for the 21st century. Unpublished.

[80] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48, New York, NY, USA, 1998. ACM.

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

[82] G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–38. ACM Press, 2004.

[83] G. Steele. *Common Lisp: The Language*. Digital Press, Newton, MA, USA, 1990.

[84] K. suk Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–88, Berkeley, CA, USA, 2002. USENIX Association.

[85] B. D. Sutter, F. Tip, and J. Dolby. Customization of Java library classes using type constraints and profile information. In *ECOOP*, pages 585–610, 2004.

[86] M. Tatsubori, S. Chiba, M. olivier Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.

[87] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

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

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

[90] T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Berlin, Heidelberg, New York, Tokyo, 1998. Springer-Verlag.

[91] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. SIAM.

[92] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[93] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01), volume 2051 of Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, 2001.

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

[95] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[96] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[97] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Not.*, 30(6):1–12, 1995.

[98] D. N. Xu, S. P. Jones, and K. Claessen. Static contract checking for Haskell. In *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages*, pages 382–399, 2007.

[99] K. C. Yeung and P. H. J. Kelly. Optimising Java RMI programs by communication restructuring. In *Middleware'03, volume 2672 of LNCS*, pages 324–343. Springer-Verlag, 2003.

Appendix A

Safe Directionality

The safe directionality property, as described in Section 4.1, is imposed on asynchronous communication models, such as Ninja', in order to preserve compatibility between peers. If both peers are in a state where both inputs and outputs are permitted, and they simultaneously send data to each other, they will both have followed different 'paths' through the session type thus risking that their respective 'believed' types for the session be incompatible. In the case of bidirectionality, we sacrifice 'path' correctness, but maintain compatibility of the believed current session types.

To see that the safe directionality property is correct for bidirectional types, we consider a session s_1 and its communicating peer s_2 such that $s_1 \bowtie s_2$. In order to derive the minimal conditions that must be imposed on s_1, we must consider the most specific s_2 such that $s_1 \bowtie s_2$; i.e. $s_2 = \overline{s_1}$. Suppose that peer p_1 of session type s_1 sends a message of type $t_1 \in \mathsf{odom}(s_1)$ simultaneously with p_2 of session type s_2 whose message is of type $t_2 \in \mathsf{odom}(s_2)$. Their session types are now respectively s_1' and s_2', where $s_1 \xrightarrow{\texttt{out}\ t_1} s_1'$ and $s_2 \xrightarrow{\texttt{out}\ t_2} s_2'$. We should now expect p_1 to be able to handle the message sent from p_2 in its new session s_1'. For this to be the case, $s_1 \leq s_1'$. Similarly, $s_2 \leq s_2'$, which may be rewritten $\overline{s_1} \leq \overline{s_1''} \Rightarrow s_1'' \leq s_1$ where $s_1 \xrightarrow{\texttt{in}\ t_2} s_1''$ by definition 4.1.5 and the standard session typing result:

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

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