Mobile Computation with Functions

Zeliha Dilsun Kırlı

Doctor of Philosophy

Laboratory for Foundations of Computer Science
Division of Informatics
University of Edinburgh

2001

Abstract

The practice of computing has reached a stage where computers are seen as parts of a global computing platform. The possibility of exploiting resources on a global scale has given rise to a new paradigm – the mobile computation paradigm – for computation in large-scale distributed networks. Languages which enable the mobility of code over the network are becoming widely used for building distributed applications.

This thesis explores distributed computation with languages which adopt functions as the main programming abstraction and support code mobility through the mobility of functions between remote sites. It aims to highlight the benefits of using languages of this family in dealing with the challenges of mobile computation. The possibility of exploiting existing static analysis techniques suggests that having functions at the core of a mobile code language is a particularly apt choice.

A range of problems which have impact on the safety, security and performance of systems are discussed here. It is shown that types extended with effects and other annotations can capture a significant amount of information about the dynamic behaviour of mobile functions and offer solutions to the problems under investigation.

The thesis presents a survey of the languages Concurrent ML, Facile and PLAN which remain loyal to the principles of the functional language ML and hence inherit its strengths in the context of concurrent and distributed computation. The languages which are defined in the subsequent chapters have their roots in these languages.

Two chapters focus on using types to statically predict whether functions are used locally or may become mobile at run-time. Types are exploited for distributed-call tracking to estimate which functions are invoked at which sites in the system. Compilers for mobile code languages would benefit from such estimates in dealing with the heterogeneity of the network nodes, in providing static profiling tools and in estimating the resource-consumption of programs. Two chapters are devoted to the use of types in controlling the flow of values in a system where users have different trust levels. The confinement of values within a specified mobility region is the subject of one of these. The other focuses on systems where values are classified with respect to their confidentiality level. The sources of undesirable flows of information are identifed and a solution based on noninterference is proposed.

Acknowledgements

I have spent five wonderful years in Edinburgh since I came to do the MSc course. I owe the most special thanks to my supervisor Stephen Gilmore for making my postgraduate studies such a pleasant experience. He introduced me to interesting research topics and guided me throughout in the most inspiring and encouraging way. I am indebted to him for reading every piece I wrote with scrutiny and for providing insightful comments. It is a rare chance to work with someone whose advice is always so helpful and so thoughtfully communicated; I am very fortunate. My gratitude for Stephen Gilmore is not only for his invaluable support during my studies but also for preparing me for the life after PhD.

I would like to thank my second supervisor Jane Hillston for providing feedback which led to improvements in many parts of this thesis. It has been comforting to know that she was keeping an eye on my progress. I also would like to thank both Stephen Gilmore and Jane Hillston for making it possible for me to attend several international meetings and to make contact with researchers outside Edinburgh.

I am very glad to have been a member of the LFCS which is an excellent research environment full of friendly people. I would like to thank Martin Hofmann and David Aspinall for driving me into an interesting project in my final year and for being tolerant as I devoted a significant amount of time to writing this thesis. Thanks are also due to my first ever collaborator Chris Walton whose ideas and cooperative spirit let me make an exciting start to research. I appreciate the encouragement I received from Don Sannella and Ian Stark. I also thank Bonnie Webber for her guidance.

I am grateful to all of my friends for not allowing any thought of loneliness to cross my mind. I will remember the nice company of Matías Menni and the chats with him and Sibylle Fröschle which were an unmissable part of my everyday life.

I would like to thank Murat Kaynar for keeping me upbeat with his contagious joy of life and making it possible for me to work with peace of mind at the same time.

I have always drawn a lot of strength from my parents' love and unfailing support. I thank them dearly for being the generous and understanding people they are.

At various stages in my research, The British Council, Türk Eğitim Vakfı, the University of Edinburgh and EPSRC provided financial support for which I feel indebted.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Some parts of this work are based on previously published papers [Kır99a, Kır99b, Kır00, Kır01].

(*Zeliha Dilsun Kırlı*)

To my parents İnciser and Orhan Kırlı

Table of Contents

1 Introduction **1**

1.1 Mobile computation with functions 1

1.2 Type and effect based static analysis 2

1.3 Overview of the thesis 4

2 Towards Mobile Functions **6**

2.1 Concurrent and distributed computation 7

2.1.1 Concurrency 7

2.1.2 Distribution and mobility 9

2.1.3 Safety and security 14

2.2 ML with concurrency and distribution 17

2.2.1 Concurrent ML 17

2.2.2 Facile 21

2.2.3 Packet Language for Active Networks 25

2.2.4 Conclusions 30

2.3 A Core language for mobile code 31

2.3.1 Aims and approach 31

2.3.2 The Core Language 31

2.3.3 Evaluation rules 32

2.3.4 Type system 34

3 Estimating Mobile Values **38**

3.1 Application areas 39

3.1.1 Compiler optimizations 39

3.1.2 Cost profiling 40
3.2 Potential mobility 41
3.2.1 Mobile functions 41
3.2.2 Mobile channels 43
3.2.3 Related work 43
3.3 Mobile-λ 44
3.3.1 Abstract syntax 44
3.3.2 Dynamic semantics 45
3.4 Type system 52
3.4.1 Semantic objects 52
3.4.2 Typing rules 55
3.4.3 Examples 57
3.5 Formal properties of the type system 59
3.5.1 Types for intermediate expressions 60
3.5.2 Type soundness 61
3.5.3 Principal typing 64
3.6 Static estimation 65
3.6.1 Extracting labels 65
3.6.2 Soundness 66
3.7 Concluding Remarks 67

4 Distributed Call-Tracking **69**
4.1 Security through language restrictions 70
4.1.1 Termination and resource bounds 70
4.1.2 Isolation and strong typing 71
4.1.3 Exploiting static analysis 72
4.2 rEval-λ 72
4.2.1 Abstract syntax 73
4.2.2 Dynamic semantics 73
4.2.3 Examples 76
4.3 A Monomorphic type system 78
4.3.1 Semantic objects 79

4.3.2 Typing rules 79
4.3.3 Examples 81
4.3.4 Formal properties 84
4.4 A Polymorphic type system 88
4.4.1 Semantic objects 88
4.4.2 Typing rules 89
4.4.3 Examples 90
4.4.4 Formal properties 93
4.5 Concluding remarks 94

5 Confined Mobile Functions **97**
5.1 Why restrict mobility? 98
5.2 Computing with mobility regions 99
5.2.1 System model 99
5.2.2 Mobility regions 99
5.3 Confined-λ 100
5.3.1 Abstract syntax 101
5.3.2 Dynamic semantics 101
5.3.3 Examples 102
5.4 Type system 106
5.4.1 Semantic objects 106
5.4.2 Typing rules 107
5.5 Formal properties 110
5.5.1 Confinement in a mobility region 110
5.5.2 Strong confinement 111
5.6 Related work 112
5.7 Concluding remarks 116

6 Noninterference and Mobile Functions **118**
6.1 Noninterference 119
6.1.1 A general characterization 119
6.1.2 A restriction on the input/output relation 120

6.1.3 Closer look at Mobile-λ . . . 120
6.1.4 Conditional expressions . . . 121
6.1.5 Example . . . 121
6.2 Secure Mobile-λ . . . 123
6.2.1 Abstract syntax . . . 123
6.2.2 Dynamic semantics . . . 124
6.3 Type system . . . 128
6.3.1 Semantic objects . . . 128
6.3.2 Typing rules . . . 129
6.4 Formal properties . . . 132
6.4.1 Consistency . . . 133
6.4.2 Noninterference . . . 136
6.5 Concluding remarks . . . 139

7 Conclusions **142**
7.1 Natural support for code mobility . . . 142
7.2 Type systems and security . . . 143
7.3 Further work . . . 144

A Selected Proof Cases **145**
A.1 Selected proof cases from Chapter 3 . . . 145
A.2 Selected proof cases from Chapter 4 . . . 147
A.3 Selected proof cases from Chapter 5 . . . 150

Bibliography **153**

Chapter 1

Introduction

1.1 Mobile computation with functions

The recent developments in telecommunications technology have made it possible to envisage a global computing platform in which computers interact easily and share a wide range of resources. Computers are no longer viewed as largely self-contained computing devices which use local resources and occasionally communicate with each other. The traditional assumptions about computation in distributed systems and desirable features for programming languages are being revised to allow for better use of the global infrastructure. A consequence of this has been the emergence of *the mobile computation paradigm* along with its supporting technologies. The key characteristic of this paradigm is to give programmers control over the mobility of code or active computations across the network by providing appropriate language features. Therefore, a typical mobile computation language is expected to facilitate the expression and execution of mobile code-containing entities. The dynamism and flexibility offered by this form of computation, however, brings about a set of problems, the most challenging of which are relevant to safety and security.

Opinions are diverse as to the primary concerns of languages for mobile computation. We argue that a sound formal foundation is of the greatest significance. By a formal foundation we mean a collective body of work which describes the computational model of the language at a suitable level of abstraction and enables rigorous or even formal reasoning about programs. Such a foundation would preclude ambiguities about the meaning of programs while also enabling the formulation and proof of certain properties including safety and security related ones.

Functional languages are known for their well-understood computational models and their amenability to formal reasoning. They also have strong expressive power due to higher-order features. Functions can flow from one program point to another as first-class values. These facts suggest that the kind of mobile computation language we put forward can be obtained by adopting a functional core and extending it with features which remain loyal to the principles of functional computation. In such a language functions can represent mobile code-containing entities and formal systems for reasoning about functional programs can be further exploited to reason about the behaviour of mobile code.

In general, this thesis contributes simple but inspiring ideas to the research in formal models of mobile computation and program analysis. In particular, novel applications of type and effect based analysis and suggestions for future directions are presented.

1.2 Type and effect based static analysis

Conventionally type systems for functional languages have been used to ensure that programs cannot corrupt the run-time representation of data values so that further execution of the program is not faithful to the language semantics. This property is known as *type safety* in the literature. Effect systems were initially proposed as a solution to the problems encountered in preserving type safety and polymorphism while integrating functional and imperative features. The basic idea was to enhance the type systems so that the expressions were associated with their observable side-effects as well as types and to use this information in making judgements with respect to safety. Some authors have further explored the use of type and effect systems for memory management and safe integration of concurrent and functional features.

The exploitation of type and effect systems need not be confined to the enforcement of type safety. Annotated with effects and other kinds of information, types can capture a significant amount of static information about a program's potential dynamic behaviour. The general methodology of type and effect systems then consists of devising a semantics for the language, expressing a program analysis by means of types and effects and showing the semantic correctness of this analysis. In other words, the type system extracts the overall behaviour of the program as a first step and as a later step one can devise various analyses to reason about it in a sound way. These analyses may be put to use in various areas such as compiler optimisations, cost profiling and safety and security. The literature includes examples of such analyses devised prior to the emergence of the mobile computation paradigm. This work introduces new analyses motivated by the characteristics of mobile computation.

A slightly different approach to exploiting type and effect systems can be to determine the properties which are desirable for all programs and design the type and effect system so that those programs which violate these properties are rejected by the system. This is closer in spirit to the earlier exploitations of type and effect systems for enforcing type safety. In the context of mobile computation, enforcing type safety alone is not sufficient to address many of the safety and security concerns. Just as the languages are revisited to examine their position with respect to the new paradigm of mobile computation, type and effect systems need to be revisited to adapt their methodology to the requirements of the context of mobile computation. The work presented in this thesis can be considered as a step in this direction.

Enforcing safety and security properties by type systems is an active research area where the significance of secure flow of information is emphasised. Most of the existing work is in the framework of computational models different to the one considered here. In this respect, we contribute to the area of type-based approaches to security by presenting type and effect systems which incorporate a machinery for tracing the flow of values in a distributed setting where functions are the essential elements of computation.

1.3 Overview of the thesis

Chapter 1 introduces the characteristics of mobile computation and functional computation. It argues that integrating these two paradigms can offer solutions to the problems which have proved to be challenging in the context of mobile computation. The useful role which can be played by type and effect systems is discussed.

Chapter 2 gives an overview of the process calculi which provide formal models of distributed and mobile computation. This is followed by a closer look at the programming languages Concurrent ML, Facile and PLAN (Programming Language for Active Networks). These languages point to a consistent effort to benefit from the fundamental ideas behind ML in designing and implementing languages for concurrent and distributed computation.

Chapter 3 focuses on a language similar to Facile where values of all types, including functions and communication channels, can be transmitted between remote sites. The problem investigated in this chapter is the static estimation of functions and channels which may become mobile at run-time. A static analysis such as the one considered in this chapter would be a useful asset for compilers in dealing with the heterogeneity of the network nodes, detecting the locality of certain values and providing static profiling tools.

Chapter 4 focuses on the language PLAN. The form of support for code mobility in PLAN is different from that of Facile. It is based on a remote evaluation facility for functions. The design of PLAN has been influenced by the need to meet the strong safety and security requirements of active networks; especially by the need to protect against denial of service. The subject of this chapter is distributed call-tracking by means of a type and effect system in the framework of a PLAN-like language. It is argued that for an applicative language distributed call-tracking can provide the basis for static estimation of resource consumption.

Chapter 5 shifts the focus back to a language which resembles Facile. Some distributed systems are characterized by their heterogeneity in terms of the nature of the computing devices, security requirements of the information flowing in the system and the trust level of the users. Programmers who provide code for such systems would find it useful to have a language mechanism which enables them to confine the flow of certain values to a particular part of the system – a mobility region. This chapter discusses how a static type system can be used to enforce confinement in a specified mobility region.

Chapter 6 revisits the language of Chapter 3 and introduces a variant of it where the values of the language are classified with respect to their confidentiality level. As in Chapter 5, it is assumed that users which interact with the system may not be equally trustworthy. The sources of undesirable information flows are identified and a secure information flow property based on noninterference is introduced. Programs which are accepted by the proposed type and effect system for the language are shown to enjoy this property.

Chapter 7 includes a summary of the thesis which clarifies contributions made to the research areas of functional and mobile code languages, annotated type and effect systems and the language-based approach to security.

Chapter 2

Towards Mobile Functions

The major sources of inspiration for our subject come from the research areas of functional programming and foundational models of mobile computation. The aim of this chapter is to give an overview of the existing works in these areas which provide the background to this thesis.

The idea of integrating the functional programming paradigm with other paradigms is not new. It has already given rise to the design and implementation of several languages. The language Standard ML (SML) [MTHM97] constitutes a good example for the systematic integration of functional and imperative features. Concurrent ML [Rep92] and Concurrent Haskell [JGF96] are examples for concurrent functional languages; they extend the languages ML and Haskell with a concurrent programming model. Functional languages which support distributed programming include general-purpose languages such as Facile [TLP$^+$93, Kna95], Erlang [AWWR93], the Join-calculus language [FM97], Poly/ML [Mat97] and MobileML [HY00], and domain-specific languages such as the Programming Language for Active Networks (PLAN) [HKM$^+$98]. Although the motivations and the intended application domains for these languages vary, they all share the common goal of exploiting the strengths of the functional paradigm within their application domains.

In this chapter, we take a closer look at the three of the descendants of Standard ML, namely Concurrent ML, Facile and PLAN. These languages point to a consistent effort in the functional language community to benefit from the fundamental ideas behind ML in designing and implementing languages for concurrent and distributed computation. There is a large body of theoretical work on these languages. Our aim is to contribute to this body of work by investigating how principled language design, and static type systems can help to deal with the challenges of mobile computation.

This chapter also presents a survey of the most frequently cited process calculi in the foundational study of mobile computation. The developments in the process calculi framework, particularly those which involve static type systems, are of interest to our work. This is mainly because they provide abstract and clear formulations of many interesting problems which can be dealt with using an approach based on types.

2.1 Concurrent and distributed computation

This section introduces some basic concepts and programming language design issues related to computation in large-scale distributed networks. We focus on the set of programming language design issues which we consider to be most relevant to our work. The discussion on each programming language issue is followed by a survey of the related foundational models. This section also provides a guide to the terminology used throughout the thesis.

2.1.1 Concurrency

Concurrent computation is the form of computation which consists of independent threads of control. In the presence of multiple processors, decomposing computation into independent threads of control allows different parts of a single task to be executed in parallel. This does not, however, mean that the use of concurrency is limited to systems with multiple processors. In the case of a single processor, concurrency can serve as a useful conceptual tool for structuring computation into independently executable parts whose computational steps can be interleaved. Many interactive applications utilize this style of concurrency. We should also note that concurrency arises naturally in the case of distributed computation. The different nodes of a network can be used to carry out different tasks in parallel.

2.1.1.1 Processes and communication

We refer to the threads of control which comprise a concurrent computation as *processes*. In general, we use the term "process" in a rather abstract sense as it is used in foundational models of concurrency and higher-level concurrent programming languages. However, if one considers their implementation, it would be appropriate to regard them as light-weight threads. For example, several processes of a concurrent programming language can be implemented within a single heavy-weight operating system process.

A significant design issue for concurrent programming languages is the specification and creation of processes. A language can require the set of processes to be fixed statically or it can provide a feature to enable their dynamic creation.

A large class of applications which benefit from concurrency requires a means for *communication* and *synchronization* between processes to facilitate data exchange and coordination. Shared-memory languages use a mutable shared state to implement process communication and provide mechanisms such as semaphores and monitors to prevent processes from interfering with each other. On the other hand, distributed-memory languages use message-passing primitives and provide a unified mechanism for communication and synchronization.

The scheme adopted for naming the end points of communication and the degree of synchronization are among the most important issues which characterize message-passing primitives. Throughout the thesis, we use the term *synchronous* to describe the form of communication where the sender of a message blocks until the message is received. The form of communication where the sender can continue its execution after sending the message is called *asynchronous*.

2.1.1.2 Calculi for concurrency

Seminal formalisms of concurrency such as CSP [Hoa85] and CCS [Mil89] consider static connectivity between processes. They provide an abstract model of computation where the basic resources are communication channels and the basic computation is carried out by matching input/output actions on these channels. Processes are constructed from basic actions by using combinators such as those for sequencing, parallel composition and choice.

The π-calculus [MPW92] followed these formalisms by offering a richer model. This rich model relies on the basic notion of naming and communication of these names between processes. Names can be understood as names of communication channels. In the π-calculus one can express the dynamic creation of a new name with a given scope. The fact that processes can exchange names facilitates the expression of dynamic changes in the interaction capabilities of a process with its environment. This is one of the senses in which the term "mobility" is used in the literature; the π-calculus is often described as a calculus for mobility. The π-calculus is a candidate for being the canonical calculus for concurrent computation with its expressive power and relatively tractable semantic theory. Since its first presentation several variants of the π-calculus have been proposed such as [HT91, San92, Bou97] and their behavioural properties have been investigated.

2.1.2 Distribution and mobility

The physical distribution of processes among different nodes of a network opens the way for network-wide sharing of processing power and other computational resources. Well-designed systems can exploit distributed computing facilities to improve efficiency. Some applications such as those for distributed information retrieval or teleconferencing are distributed by nature. These types of applications can be accommodated only if computation can be distributed across the system. Distribution is also essential in providing reliable and fault-tolerant services.

The issues concerning concurrency are applicable to distributed systems with the additional complexity of taking into consideration the different physical sites of computation. For example, communication between remote sites is vulnerable to link and network failures. Moreover, the nodes of a distributed system may be heterogeneous and exchanging data between these nodes may consequently require support for *interoperability*.

The distribution of processes among different physical locations makes *locality* an important notion to be addressed by language designers. The traditional approach to distributed computing focuses on hiding the presence of different localities and providing a uniform computational environment for programs. This implies that support for *transparency* is a design goal for distributed programming languages.

2.1.2.1 Mobility

In recent years many researchers have highlighted the need to revise the basic assumptions about distributed computing. Location transparency is one of the principles which is being questioned. Some researchers argue that this principle sets an obstacle to exploiting the computational infrastructure made available by the recent technological developments. Languages which enable programmers to have control over the *mobility* of code are accepted by many as better suited for computation in modern large-scale networks than languages which do not provide this control.

In designing a language which supports mobility, an crucial issue is to determine what should be allowed to be mobile. The design decision on this issue has a significant influence on the expressiveness of the language and on more practical aspects such as the feasibility of implementation. A comprehensive survey of different forms of mobility can be found in [FPV98]. According to the classification which is presented there, the form of code mobility which involves no migration of execution state is called *weak mobility*. The form of mobility which supports the transfer of both code and execution state is referred to as *strong mobility*.

Data space management is another issue which needs to be resolved by language designers. When a computational unit moves to a new computational environment, the set of bindings to resources accessible by it must be rearranged. The method of achieving this depends on the nature of the resources involved and the type of binding to these resources. A survey of approaches to data space management can be found in [FPV98]. A similar survey has been conducted in [SY97].

2.1.2.2 Calculi for distributed computation

In Section 2.1.1.2 we introduced some of the calculi for concurrency. The main topic of interest for those calculi is provide a simple and general model for studying the behaviour of concurrent systems by using algebraic methods. A topic not addressed is the physical distribution of processes among different sites of computation. In order to address the issue of physical distribution, many authors have adopted the approach of basing their work on a variant of the π-calculus. In this way, they exploit the basic powerful notions of the π-calculus. Another advantage of this approach is that it gives the opportunity to relate the theoretical results obtained by different studies in the common framework of the π-calculus. The survey paper by Hennessy includes a comparison of several location calculi [Hen98]. We include below a brief introduction to some of these calculi to note the state-of-the-art in this field.

π_l: A line of work on a distributed variant of the π-calculus was initiated by the presentation of the π_l-calculus by Amadio and Prasad [AP94]. This work focused on the notions of locality and failure for the programming language Facile. A π_l program consists of a number of processes running on one or more locations where the number of locations can dynamically change due to the generation or failure of new nodes. Processes can move between different nodes. The calculus makes clear the dependence of channels and processes on the nodes where they reside. The authors take the view that the distribution of processes can be perceived by the absence of certain communication capabilities due to failures. The π_{1l}-calculus [Ama00] which is derived from the π_l-calculus offers a model of asynchronously communicating distributed processes where every channel name is associated with a unique process.

Dπ: The language Dπ presented by Hennessy and Riely [HR98b] is a distributed variant of the π-calculus. It is different from the π_l-calculus and its extensions in two major respects. It ignores location failures and requires communication to be local. By local we mean that two processes can communicate on a channel only if both of the processes and the channel are co-located. According to the classification presented in [Hen98] Dπ can express the global migration of passive code. A calculus of distributed higher-order processes which is related to Dπ has been presented in [YH99]. In this calculus parameterized processes as well as basic values can be transferred between distinct locations. However, the distributed fragment of this calculus, is not as expressive as Dπ. This is mainly because locations do not have the first-class status as they do in Dπ.

Join-Calculus: The original work on the join-calculus of Fournet and Gonthier [FG96, FGMR96] was motivated by the identification of a gap between the theory and the practice of concurrent computation in distributed systems. According to the authors, the existing calculi provided elegant theories but they overlooked implementation issues. On the other hand, the large set of constructs found in programming languages for building concurrent distributed applications constituted an obstacle to their theoretical investigation. The design of the join-calculus was an attempt to provide a simple formal model of concurrent, distributed computation which could also be used as the foundation for a practical programming language suitable for computation in modern networks. The join-calculus can be regarded as an asynchronous variant of the π-calculus where scope restriction, reception and replication is merged in a single construct called *definition*.

The development of the Distributed join-calculus had several stages. The conceptual model [FG96] was obtained by an extension of the generic model of the chemical abstract machine [BG92]. In a later work the join-calculus was extended with explicit locations and primitives for mobility [FGMR96]. The resulting Distributed join-calculus allows the expression of mobile agents moving between different physical sites. It supports the global migration of active code. This corresponds to strong mobility in our terminology. A location resides on a physical site and contains a group of processes. It can also be moved to another site taking all of its sub-locations with it. Join locations can be organised into a tree structure. This feature of the join-calculus offers a simple model of failure. An experimental high-level language based on the calculus with the same name has been implemented in Objective CAML [Ler97]. The join-calculus has also led to the implementation of the JoCaml system which extends Objective CAML with the distributed programming model of the join-calculus.

Distributed π-calculus: Distributed π-calculus of Sewell [Sew98] combines the location and migration primitives from the Distributed join-calculus with asynchronous communication in the π-calculus style.

UDP Calculus: The recent work by Serjantov, Sewell and Wansbrough on the UDP calculus [SSW01] focuses on providing a rigorous understanding of the sockets interface to UDP (User Datagram Protocol). The authors point out that an experimentally validated operational model for socket programming would be useful in reasoning about the behaviour of practical distributed programs which use protocols such as UDP.

Ambient Calculus: The Ambient Calculus of Cardelli and Gordon [CG98, Car99] is a process calculus which is different in spirit to the π-calculus model of computation. It focuses on process mobility rather than process communication.

The abstraction of ambient is what gives the calculus its distinctive character. An ambient is a named location which may contain local processes and subambients. It can move as a unit in to or out of other ambients. The Ambient calculus can be considered as supporting local migration of active code [Hen98].

Ambients can model the existence of different administrative domains in large-scale networks. The ability of processes to cross the barriers between these domains can be expressed by the capabilities associated with the processes.

Seal (σ) Calculus: The σ-calculus of Vitek and Castagna [CV99] shares goals with the Ambient Calculus. It extends the π-calculus with location mobility and resource access control. Security issues have been emphasized in the design of the σ-calculus to allow context-independent proofs of security. The authors describe their aim as providing a model for secure distributed applications over large-scale open networks such as the Internet.

Seals are named, hierarchically-structured locations. A seal can contain a hierarchy of subseals. Communication occurs synchronously over the channels and is restricted to be either local or neighbourly. Seals may be moved over channels, this makes the Seal-calculus a higher-order calculus. The mobility of a seal is under the control of its environment. There are mechanisms to control the propagation of the names of channels in order to control external access to local resources. The notion of *portal* is proposed as the key mechanism to control inter-seal reactions.

2.1.3 Safety and security

A major motivation for mobile computation is to make better use of the global computation infrastructure by facilitating the sharing of its resources among mobile computational entities. In order to bring about the desired advantages of mobile computation, safety and security must be taken into consideration with particular emphasis. In general, safety aims at the prevention of unintended behaviour of programs and is a precondition for security. Security is concerned with a wider range of issues such as secrecy and integrity of the information which flow within a system and the prevention of malicious attacks.

Programming languages may benefit from a wide range of mechanisms to improve the safety and security of systems. Exploiting language-theoretic techniques is one of the approaches adopted by researchers and this is also the one in which we are interested. This approach focuses on principled language design where support for security is included as one of the design goals. The major challenges faced by the designers of such languages are identifying what is considered to be harmful behaviour for programs and devising mechanisms for restricting the execution of programs which are potentially harmful. Type systems appear to be useful in this context. Languages may choose to use static typing, dynamic typing or a combination of the two.

It is important to note that the mobile computation paradigm poses challenges to safety and security of systems which are hard to handle by language-based techniques alone. Incorporating mechanisms based on cryptography can become a necessity.

2.1.3.1 Security and process calculi

In the previous sections we introduced a representative set of process calculi which appear to drive much of the foundational research on mobile computation. Security is a significant topic of interest in mobile computation. Therefore, it is being widely studied within the framework of these process calculi. We will focus here on the most recent works which propose type systems to enforce certain behavioural properties concerning security.

π-calculus: A pioneering work on the use of types for enforcing secure information flow is due to Abadi [Aba99]. In this work a notion of behavioural equivalence called testing equivalence is used to formulate a secrecy property for the spi-calculus [AG99] processes. The spi-calculus is an extension of the π-calculus with cryptographic primitives. It is suitable for describing and analysing security protocols. Secret information can be manipulated by the encryption and decryption primitives throughout the computation. The role of the type system is to check statically whether a process has the desired secrecy property. Well-typed processes cannot leak secret information to the environment.

Hennessy and Riely have studied a type system for an asynchronous variant of the π-calculus [HR00] where specific security levels are assigned to input/output capabilities and processes. The type system guarantees that a process cannot access resources of a higher security level than that of itself. Additionally, in a well-typed system the behaviour of low-level processes cannot be affected by changes to the high-level behaviour. To define this formally they use an appropriate notion of testing equivalence.

A recent work by Honda, Vasconcelos and Yoshida [HVY00] presents a sophisticated type system for another variant of the π-calculus. They use input/output types annotated with security levels. Their motivation is to provide a foundational calculus into which typed programming languages can be embedded. This allows the behavioural analysis of higher-level programs with respect to secure information flow.

Another work on a security type system for the π-calculus is by Cardelli, Ghelli and Gordon [CGG00]. A primitive is added to the π-calculus for dynamic group creation with a given scope. The type system guarantees that channels created within the scope of a particular group cannot be leaked to processes outside the initial scope of the group.

Dπ: The language Dπ has played a central role in the study of type systems for mobile code security. In [HR98b] Hennessy and Riely propose a static type system for Dπ where location types are used to express the capabilities which mobile code has at a particular location. The type system guarantees that mobile code can access a resource only if it has the required capabilities. The same authors focus on security issues for open systems in [HR98c, HR99]. For such systems, one cannot rely on type-checking the whole system statically. The authors investigate partially-typed semantics for Dπ and propose a mixture of static and dynamic type-checking. Mobile code which comes from an unknown or an untrusted source is subjected to some dynamic checks.

Join-Calculus: A polymorphic typing discipline akin to the Damas-Milner typing discipline of ML has been developed for the join-calculus [FLMR97]. Although the join-calculus provides the framework for investigating distributed systems security in a number of works [AFG98, AFG99], there do not appear to be any attempts to exploit type systems for this purpose.

Ambient Calculus: There are a large number of type systems proposed for the Ambient calculus which deal with different aspects of security in the computational model induced by mobile ambients [CG99, CGG99b, CGG99a, LS00, BC01]. The type system of [CG99] is designed to prevent run-time errors caused by the incompatibility of the types of exchanged messages. The type system of [CGG99b] distinguishes between mobile and immobile values. The work presented in [CGG99a] extends the Ambient calculus with a group creation primitive. This type system can exploit groups to identify the set of ambients that a process may cross or open.

Levi and Sangiorgi study the possible forms of interference between mobile ambients and their impact on security in [LS00]. A new calculus called Mobile Safe Ambients is introduced. This calculus imposes restrictions on the interactions of ambients to prevent undesirable interferences. Bugliesi and Castagna build on this work and introduce Secure Safe Ambients [BC01]. Their type system can express behavioural invariants for mobile ambients. It allows the detection of security threats posed by hostile ambients which exploit implicit acquisition of capabilities to access sensitive resources. The authors also discuss how their work relates to the security architecture of the Java Virtual Machine [LY97].

2.2 ML with concurrency and distribution

2.2.1 Concurrent ML

Concurrent ML (CML) is a programming language, developed by John Reppy [Rep92]. It integrates high-level abstraction mechanisms with concurrency primitives. The successful exploitation of procedural abstraction and data abstraction has been a breakthrough in sequential programming. In his introduction to CML Reppy draws attention to this fact. At the time when CML was designed, the practice of concurrent programming had mainly been based on low-level systems programming languages which provided abstractions of hardware. The work of Reppy on CML was driven by the aim of facilitating programmer-defined abstractions for concurrent programming. His work evolved around the question *"What is the right notion of abstraction for concurrent programming"?*

2.2.1.1 Higher-order concurrency

The sequential fragment of CML is inherited directly from Standard ML. Support for dynamic process creation and interprocess communication are two of the essential extensions to this sequential fragment. Reppy claims that shared-memory communication is ill-suited for an ML-based language because it relies on mutable state and leads to an imperative programming style. He argues in favour of message-passing communication by stating that it provides a level of abstraction which is in keeping with the basic design philosophy of CML. CML processes communicate on dynamically-created channels and the communication is synchronous. Reppy justifies his choice for synchronous communication by explaining that reasoning about protocols is easier in the case of synchronous communication.

The key underlying idea of CML is to separate the operation of synchronization from the mechanism for describing synchronization and communication protocols. CML introduces a new abstract type of values called *event*. Events represent potential communication and synchronization actions. These abstracted actions are performed only when they are synchronised upon.

By the introduction of the *event* datatype synchronous operations are elevated to

Property	**Function Values**	**Event Values**
Type constructor	$\rightarrow$	event
Introduction	λ-abstraction	recvEvt, sendEvt, ...
Elimination	application	sync
Combinators	$\circ$, map, ...	choose, wrap, ...

Figure 2.1: Functions and Events

being first-class values. One can draw an analogy between synchronous operations and functions; an event being analogous to a function abstraction and synchronization being analogous to function application. CML also provides combinators for the construction of more complex events from simpler ones (see Figure 2.1).

Reppy uses the term *higher-order concurrent programming* for the style of concurrent programming promoted by CML. This style is characterised by the ability to express a wide range of concurrency paradigms by using events and a small set of primitives and combinators.

An overview of common CML operations is given in Figure 2.2. The combinators guard and wrap create events from pre-synchronization and post-synchronization actions respectively. The function f in guard f represents a suspended function whose evaluation is forced upon synchronization on the guard event. Its result is used in the synchronization. The event ev in wrap(ev,f) is wrapped with the function f. When the wrap event is synchronized on, the function f is applied to the synchronization result of event ev. It should be relatively obvious what the rest of the operations do.

2.2.1.2 Generalised selective communication

Reppy takes the view that support for generalized selective communication is essential for a concurrent programming language. He points out the limitation imposed by the notion of selective communication of CSP [Hoa85] in which only input guards are allowed. He attempts to generalise this notion so that both input and output operations are allowed as guards.

The interplay between abstraction and generalized selective communication has

```
type  thread_id
type  'a chan
type  'a event

val  spawn: (unit → unit) → thread_id

val  channel: unit → 'a chan
val  recv: 'a chan → 'a
val  send: ('a chan * 'a) → unit
val  recvEvt: 'a chan → 'a event
val  sendEvt: ('a chan * 'a) → unit event

val  guard: (unit → 'a event) → 'a event
val  wrap: ('a event * ('a → 'b)) → 'b event
val  choose: 'a event list → 'a event

val  sync: 'a event → 'a
val  select: 'a event list → 'a
```

Figure 2.2: Overview of CML operations

been a key issue influencing the design of CML. The examples in [Rep92] and [Rep99] may convince the reader that the new abstraction mechanism *event* is indeed indispensable for a realistic integration of abstraction with selective communication. It hides the details of the communication protocols while allowing the expression and implementation of selective communication.

2.2.1.3 Mobility

CML is a concurrent programming language and support for distributed programming is not among its design goals. Naturally, we cannot talk of mobile computation in the sense in which it is used throughout this thesis. However, values of CML which include channels and functions do move between processes and the communication topology evolves dynamically. We can use the term mobility in the sense in which it is used for the π-calculus. One can view CML channels and the send and recv operations as providing an implementation of a typed version of the π-calculus. The key difference is that in the π-calculus one can send free channel names along channels whereas in CML a channel has to be created in some scope before being sent. CML bears an even closer resemblance to the higher-order extensions of the π-calculus where λ-calculus terms are allowed to be exchanged over channels.

2.2.1.4 Safety and security

CML adopts static typing to enforce type safety in the style of Standard ML. The polymorphic type system of ML has been adapted to support polymorphism of channels. In his thesis Reppy presents a type safety property for a subset of CML which includes the essential concurrency extensions such as channels and events.

2.2.1.5 Formal foundations

A formal mathematical model was not one of the design goals of CML. The main motivation was to produce a realistic language to be used in large-scale computer programming. However, the algebra of events was observed to have some properties which makes it interesting from a mathematical point of view. Alan Jeffrey has investigated the categorical structure of CML and its denotational semantics [Jef96]. Some other authors have worked on the semantic foundations of CML using the frameworks of action semantics [MM94] and process calculi [FHJ96]. CML has also been extensively studied within the field of static analysis [NN96b, NN95, NN94, GFH97, BD97].

In his thesis Reppy also presents an operational semantics for a small language λ_{cv} which models the concurrency features of CML. The type safety result mentioned above has been obtained with respect to the semantics of λ_{cv}. A revised version of this semantics appears in [Rep99].

2.2.1.6 Comments

The emphasis in the design of CML is on combining programmer-defined abstractions for concurrency and generalized selective communication. A fully-developed functional mobile code language would benefit from the kind of support provided by CML for concurrent programming. However, the emphasis of our work is not on programming convenience. The influence of CML on our work is due to the following list of conclusions we have drawn.

- In a language where concurrently executing processes can communicate by exchanging values, giving first-class status to functions and channels increases the expressive power of the language.
- Synchronous message-passing communication over typed channels is well-suited for a concurrent programming language based on ML.
- The principle of strong typing can be adapted to concurrent programming.
- Existing implementations of ML such as SML/NJ [AM91] can be exploited in implementing languages which support concurrent programming.

2.2.2 Facile

Facile [TLP+93] is a language which aims to encompass functional, imperative, concurrent, and distributed programming paradigms in a single programming language. The original work on Facile focused on the formal foundations of the functional, concurrent language integration and on abstract implementation models [GMP89]. This work was influenced by the work on process calculi such as CCS [Mil89] and CHOCS [Tho89]. It was investigated further by Knabe to support the mobile computation paradigm [Kna95].

2.2.2.1 Integration of paradigms

A major principle in the design of Facile is the symmetric integration of different programming paradigms so that every paradigm can use any other paradigm as a subcomponent for its expression. For example, a function may be implemented as a system of communicating processes and the internals of a process may be implemented using functions.

The designers of Facile emphasize the importance of simplicity and coherence of concepts and language constructs. The number of concepts and constructs must be relatively few. They must be easy to understand and their meaning must not be too sensitive to their context. Except for a few which involve behaviours most of the language constructs can be expressed in the spirit of the λ-calculus using function application and values.

Facile adopts the principle of uniform treatment of values from Standard ML. All values are treated equally. For example, scripts, channels, guards, nodes and libraries are all first class values. This principle enables Facile to inherit many of the benefits of Standard ML as well as facilitating the implementation of applications which require dynamic connectivity.

2.2.2.2 Concurrency

Facile's model of computation depends on multiple concurrently executing processes. Processes can be created dynamically and they execute by evaluating expressions. The behaviour of a process is syntactically described by a behaviour expression. Essentially, behaviour expressions are not expressions because they do not denote values. The simplest behaviour expression is terminate which denotes a dead process. The other basic form of behaviour expression is activate exp where exp evaluates to a process script. A script can be thought of as the code executed by a process. The language provides constructs, script and activate for converting a behaviour expression into a script and vice versa. Behaviour expressions also include parallel composition of behaviour expressions and non-deterministic choice. Processes communicate over synchronous channels. Any value which can be defined in the language can also be communicated over channels.

We choose to give an overview of Facile by means of an example. The basic operations are similar to those of CML except for the characteristic event synchronization mechanism described in Section 2.2.1. Figures 2.3 and 2.4 show different implementations of the same function illustrating the fact that Facile can support different programming approaches.

```
proc fib_server(a,b) = let fun fib(i) = if (i = 0) or (i = 1) then 1
                                         else fib(i-1) + fib(i-2)
                       in b ! (fib(a?))
                       end;
                       terminate
```

Figure 2.3: Processes use functions

```
proc fib_server(a,b) =
      let fun fib(i) = if (i = 0) or (i = 1) then 1
                       else
                          let val (in1,out1) = (channel(int), channel(int));
                              val (in2,out2) = (channel(int), channel(int));
                          in
                              spawn(out1 ! (fib(in1 ?)); terminate);
                              spawn(out2 ! (fib(in2 ?)); terminate);
                              (in1 ! (i-1));
                              (in2 ! (i-2));
                              ((out1 ?) + (out2 ?))
                          end
      in b ! (fib(a?))
      end;
      terminate
```

Figure 2.4: Functions use processes

2.2.2.3 Distribution and mobility

To address the locality of processes the notion of *node* has been introduced. A Facile system can be viewed as a collection of nodes each of which host a number of processes. A node corresponds to a virtual processor with an address space. Nodes can be created dynamically and may reside on different computers in a network. The language also provides the constructs r_spawn and r_channel to create processes and channels at specific nodes.

Since the implementation of the choice operator of CCS leads to problems in a real distributed setting, Facile adopts a different version of the choice operator which is discussed in detail in [GMP89]. Facile also provides some general constructs to implement delay and time-out mechanisms to circumvent the problems posed by blocked communications.

The fact that functions are first-class values means that we can create functions at run-time, apply them to arguments, pass them to other functions as arguments and receive them as results. We can also transmit them over communication channels. All of these properties imply that mobile agents have a natural representation as functions in Facile.

However, there are other requirements for Facile to be generally accepted as a mobile computation language for the global computing platform. One such requirement is the ability to deal with heterogeneity of network nodes. Different nodes may be of different architectures and therefore support different value representations. Knabe has demonstrated different approaches for dealing with this issue and implemented a language which can be classified as a weakly mobile language [Kna95]. This language combines strong typing, remote resource access and independent compilation which are desirable properties for a language for mobile computation.

2.2.2.4 Safety and security

Facile adopts static typing in the style of ML. The original definition of Facile presents a monomorphic type system for Facile. The authors have also presented a polymorphic type and effect system for a subset of Facile along with a type inference algorithm [Tho94].

2.2.2.5 Formal foundations

A clean and well-understood semantics has been the main motivation from the very early days of Facile. This has led to a number of works on the formal foundations of Facile such as [GMP89, LT95, AP94]. It continues to be of interest to researchers of process calculi.

2.2.2.6 Comments

Our study of the language Facile reinforces the ideas we formed as a result of our study of CML. We focus on Facile more closely in our work because it supports distributed computation. It is also the case that language support for mobile computation has previously been investigated within the framework of the Facile project. However, this work has an emphasis on practical issues, whereas our work has a more theoretical slant. Moreover, we are not constrained by any specific language infrastructure to build upon.

2.2.3 Packet Language for Active Networks

Packet Language for Active Networks (PLAN) [HKM+98] is a domain-specific, simple functional language for programs which form the packets of an active network. It is based on a subset of ML with some primitives to express remote evaluation. PLAN is being developed at the University of Pennsylvania as a part of the SwitchWare project [AAH+99] which is one of the prominent projects within the area of active network research [TSS+97].

2.2.3.1 Active Networks

The concept of active networking has been motivated by the desire to bring programmability to networks. Active networks are *active* in the sense that switches perform customized computations on the packets flowing through them. This can be contrasted with the approach adopted in traditional networks. In these networks the nodes transport data passively; computation is limited to header processing for packet-switching networks and signalling for connection-oriented networks. The principal advantages of active networking are to be seen in the enabling of adaptive protocols, implementation of application-specific functions at strategic points within the network and the deployment of new services at a faster pace [TSS+97].

The SwitchWare project explores how to make the network programmable by allowing switches to be dynamically extended with new services and by allowing packets themselves to be programs. The idea of packets as programs is being explored through the design and implementation of the PLAN language.

The SwitchWare architecture is based on three layers. The top layer consists of active packets which are mobile entities containing both code and data which replace the header and payload of conventional packets. The middle layer consists of extensions which may be dynamically loaded or which can be part of the basic functionality of a switch. The lowest layer is static and provides a secure foundation for the layers above itself.

The active packet layer is intended for high-level control while the complex functionality resides in the services which are provided by the middle layer. Thus PLAN was designed to support lightweight programmability for packets while also providing a scripting language for general services which may employ heavyweight computations. The most recent implementation of PLAN has been carried out in Objective CAML [Ler97].

2.2.3.2 Concurrency and distribution

As a language designed specifically for active networks, PLAN supports concurrent and distributed execution of programs carried in active packets. It is a purely functional language and its packets do not communicate with each other. This ensures non-interference among concurrently executing programs. Service layer extensions may be written in other general-purpose languages. This can introduce possibilities for communication.

A PLAN application consists of a series of PLAN packets which comprise a task. A host application constructs a PLAN packet and injects it into the active network through a port connected to the local PLAN interpreter.

Packets: A PLAN packet encapsulates a *chunk (code hunk)* and the fields *evaluation destination, resource bound, routing function name*, *source* and *handler*.

<table>
<tr><th colspan="8">Packet</th></tr>
<tr><td colspan="3">chunk</td><td rowspan="2">evalDest</td><td rowspan="2">RB</td><td rowspan="2">routFun</td><td rowspan="2">source</td><td rowspan="2">handler</td></tr>
<tr><td>code</td><td>entry point</td><td>bindings</td></tr>
</table>

A chunk is composed of three components: PLAN code, a function name to serve as an entry point and values to serve as bindings for the function's arguments. Chunks are first-class data values and their execution can be forced by the core service eval.

The code consists of a series of definitions which bind names to functions, values and exceptions where the names of the services available at the node of definition form the initial bindings in the namespace. The arguments are evaluated locally in a call-by-value fashion and the actual evaluation of the function call is delayed until the packet arrives at its destination. The function call takes place in an environment where all top-level bindings are available. This is the point where PLAN departs from the discipline of static scoping which it adopts elsewhere.

The roles of the remaining fields of a packet are as follows. The routing function serves to define how the packet will be transported from the current node to the evaluation destination. The resource bound sets the limit on the number of hops the packet or any of its descendants can make. This restricts the global network resource usage of a PLAN application. The source field names the packet's oldest ancestor and the handler field provides the name of a service routine on the source which will handle certain communication errors.

Network Primitives: PLAN programs create new packets through calls to the network primitives OnRemote and OnNeighbor. The call OnRemote(C,evalDest, Rb, routFun) creates a new packet which will evaluate chunk C on node evalDest. As we have noted above, the bindings of the chunk are determined locally while the function application is evaluated remotely. The arguments Rb and routFun correspond respectively to the resource bound and the routing function name fields of the created packet. Until it reaches its evaluation destination, Rb is decremented by one at each hop and the packet is terminated if the resource bound is exhausted. The network primitive OnNeighbor is similar to OnRemote the difference being that the created packet must execute on a neighbour of the current node.

Services: PLAN programs can call core services which are present on all active nodes in the same way as they call locally-defined functions. Core services are guaranteed to terminate. The services thisHost, getHostByName, getNeighbors, getRB, defaultRoute, print are examples of core services presented in the Specification of PLAN [KHMG99] to be provided as standard library functions. In addition to these, there are a number of service packages which extend the functionality of PLAN programs. For example, the service package resident enables PLAN programs to leave data on the nodes they visit to facilitate exploring the network topology. Note that service packages such as these may create the possibility for unsafe operations and therefore PLAN may have to impose certain safety and security requirements to permit their employment.

2.2.3.3 Mobility

In Section 2.2.3.2 we have introduced the execution model for PLAN programs in an active network. Mobility of PLAN packets and the remote evaluation of chunks encapsulated in these packets is at the heart of the execution of PLAN programs. Given its domain-specific approach, it is not straightforward to compare PLAN with general-purpose languages to classify the kind of mobility it supports. Nevertheless, we consider PLAN to be strongly mobile due to the fact that PLAN programs are able to initiate their own evaluation at a remote site as well as taking with them a collection of resources which they may need at that remote site.

2.2.3.4 Safety and security

PLAN has a limited set of simple constructs for flow of control. It supports statement sequencing, conditional execution, iteration over lists by folding, and exceptions in the style of ML. Recursive functions and unbounded iteration are ruled out to ensure the termination of programs. Besides these limitations on its expressiveness, we have also noted its resource-limited semantics. All of these restrictions are intended to enforce safety and security in a simple way. Indeed, pure PLAN programs, which use core services only, can run with no need for authentication.

PLAN is strongly typed which implies that well-typed programs cannot go wrong. It requires that programs are statically typeable but it also allows dynamic type checking. A discussion about the relative merits of static and dynamic type checking for PLAN can be found in [KHMG99].

The safety and security of pure PLAN programs can be ensured by the mechanisms presented above. However, PLAN programs can also call service routines which are written in general-purpose languages. This constitutes a potential threat to the safety and security of the system. To make service calls safe the pure part of PLAN has been complemented with a system of trust management [HK99]. According to this system, each node administrator creates a policy which restricts the use of unsafe services to selected users through a process of authorization. Packets are then required to authenticate themselves before accessing the privileged services. The technique employed by PLAN is called *namespace-based security*. It is based on expanding or contracting a packet's service environment depending on its level of privilege.

Controlling access to resources is an important part of providing security within a multi-user system. However, it is usually not sufficient to control the flow of information within the system. A line of research on the PLAN language focuses on developing a theory of information flow for PLAN-like languages [KGA00].

2.2.3.5 Formal Foundations

PLAN has recently been provided with a specification which aims to define a mathematically precise operational semantics [KHMG99]. It is intended to set a standard for implementations and to support proofs of the key properties of PLAN which all conformant implementations must obey.

The designers of PLAN have put emphasis on the language being sufficiently well defined so that advances in type theory, programming language semantics and formal methods can be exploited to address issues related to safety and security. It has the λ-calculus at its core and adopts many features of ML for its well-defined foundations. Hence, it is possible to benefit from the existing work in programming language theory.

2.2.3.6 Comments

A close look at PLAN makes one realize that the support for mobility in a functional language need not depend on the mobility of values over channels as is suggested by CML and Facile. Instead, a functional mobile code language can adopt a variant of the remote evaluation model introduced by PLAN if it suits its intended application domain. PLAN also draws our attention to the fact that in a system with limited resources restricting the expressiveness of the language may be a useful method for enforcing certain safety and security requirements.

2.2.4 Conclusions

Language	**CML**	**Facile**	**PLAN**
Process Creation	*dynamic*	*dynamic*	*dynamic*
Communication	*message-passing* *first-class channels* *synchronous*	*message-passing* *fist-class channels* *synchronous*	*No*
Distribution	*No distinct localities*	*distinct localities*	*distinct localities*
Mobility	*channel mobility*	*weak code mobility*	*strong code mobility*
Static Typing	✓	✓	✓
Dynamic Typing	*No*	*No*	✓

Figure 2.5: Summary

Figure 2.5 summarises the approaches taken by different language designs with respect to the issues which we have considered in detail throughout this section. Our concise reading of this picture is as follows. All of the languages we have considered support dynamic creation of concurrent computational units indicating that this is a requirement to accommodate the dynamics of modern networks. Whether to support communication or not remains an issue of choice depending on the targeted application domain of the language. If communication between processes is supported at all, synchronous message-passing communication over channels is considered to be more in accord with the philosophy of ML. Increasing demand for control over the mobility of computations can be observed by inspecting the chronological order of the developments around these languages. We should also note that the difficulty of achieving strong mobility without compromising safety or security is a commonly agreed fact. Type systems are seen as useful tools for providing safety. The guarantees offered by static typing seem to be attractive to all, however some languages may find flexibility equally important and seek to reconcile the advantages of static and dynamic typing.

2.3 A Core language for mobile code

2.3.1 Aims and approach

So far in this chapter we have looked at the state-of-the art in the foundational studies for mobile computation and the state-of-the art in ML-based language design for concurrent and distributed programming. The aim of the work presented in later chapters is neither to propose a new process calculus nor to design a fully-fledged functional mobile code language. We base our work on a series of small functional languages which are derived from those of the previous section. These languages serve as meta-languages for investigating a range of problems, such as the estimation of mobile values in a Facile-like language and call-tracking analysis for a PLAN-like language by using a type-based approach. The existing works on CML, Facile and PLAN have served as a valuable departure point for this stream of work. We have also introduced new research points for functional languages. Type systems for secure information flow have not yet been studied as extensively in the framework of functional distributed languages as in the frameworks of process calculi and imperative languages. Those parts of this work which investigate secure information flow for mobile functions have been motivated by this fact.

The meta-languages presented throughout the thesis can be considered as candidates for typed intermediate languages to be used in modern type-preserving compilers. This idea is rooted in the fact that some of the recent compilers for functional languages use higher-order typed intermediate languages which facilitate sophisticated type-based analyses [TMC$^+$96, JLM97, Sha97]. The information obtained by these analyses has so far proved to be useful in optimizations, and promoting the efficiency of both data representation and garbage collection. In the context of mobile computation, the intermediate language level appears also to be a suitable level to deal with security problems.

2.3.2 The Core Language

We now present a language – the Core – which corresponds to a common subset of the sequential fragments of CML, Facile and PLAN. It is a simple extension of the

Constants	c	$::=$	$()$	unit
			$\mid n$	integer
			$\mid \mathsf{true} \mid \mathsf{false}$	boolean
Expressions	e	$::=$	c	constant
			$\mid x$	value identifier
			$\mid \mathsf{fn}\, x \Rightarrow e$	function abstraction
			$\mid e_1\, e_2$	function application
			$\mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$	conditional
			$\mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	local binding
			$\mid e_1\ op\ e_2$	primitive operation

Figure 2.6: Abstract Syntax for the Core

λ-calculus and all of the languages which appear in the later chapters are derived from it in one way or another. Figure 2.6 gives the abstract syntax of this language. In the rest of the section we first determine a set of rules which govern the evaluation of an expression. We then present a type system which is used to judge whether an expression is well-formed according to a set of typing rules.

2.3.3 Evaluation rules

We adopt the structural approach to defining operational semantics [Plo91] where the evaluation of an expression is defined in terms of the evaluation of its subexpressions. Each sentence of the form $e \rightarrow e'$ defines one step in the evaluation so that e' is the result of the first step of evaluation of e. A rule may be an axiom in the form of a single sentence or an inference rule where the sentences above the bar represent the hypotheses and the sentence below represents the conclusion. For simplicity, we assume that the expressions are closed which means that they do not contain any free variables.

$$\textbf{Values} \qquad v \quad ::= \quad () \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{fn}\, x \Rightarrow e$$

The expressions which cannot be further evaluated are called *canonical expressions*. The unit value, integers, booleans and function abstractions are canonical expressions and denote the values of the Core.

Function application

(1) $\dfrac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}$ **(2)** $\dfrac{e_2 \longrightarrow e_2'}{v\, e_2 \longrightarrow v\, e_2'}$ **(3)** $(\mathsf{fn}\, x \Rightarrow e)\, v \longrightarrow e\{v/x\}$

The first two rules above indicate that the expressions are evaluated in left-to-right order. The third rule shows that in order for evaluation to proceed the value of the first expression must be a function closure. The notation $e\{v/x\}$ denotes the substitution of value v for variable x in expression e, where the necessary renaming is assumed to have taken place to avoid the capture of free variables. This rule also reveals the strict nature of the language. The expression in the argument position is fully evaluated before it is substituted for the formal parameter in the body of the function.

Conditional expression

(1) $\dfrac{e_1 \longrightarrow e_1'}{\mathsf{if}\ e_1\,\mathsf{then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow \mathsf{if}\ e_1'\,\mathsf{then}\ e_2\ \mathsf{else}\ e_3}$

(2) $\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow e_2$ **(3)** $\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \longrightarrow e_3$

The evaluation of a conditional starts by the evaluation of its first expression which we refer to as the guard. The value of this expression determines which one of the two alternative branches will be taken in the rest of the evaluation. If it is true the first branch is taken and the value of the conditional expression is the value of expression e_2. Otherwise, it is the evaluation of e_3 which yields the value of the conditional.

Local binding

(1) $\dfrac{e_1 \longrightarrow e_1'}{\mathsf{let}\, x = e_1\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\, x = e_1'\ \mathsf{in}\ e_2}$ **(2)** $\mathsf{let}\, x = v\ \mathsf{in}\ e_2 \longrightarrow e_2\{v/x\}$

These rules concern the evaluation of an expression with a local binding where the scope of the variable x is the expression e_2. The first expression is evaluated first and its result is substituted for x within e_2.

Primitive operation

$$\textbf{(1)}\ \frac{e_1 \longrightarrow e_1'}{e_1\ op\ e_2 \longrightarrow e_1'\ op\ e_2} \quad \textbf{(2)}\ \frac{e_2 \longrightarrow e_2'}{v\ op\ e_2 \longrightarrow v\ op\ e_2'} \quad \textbf{(3)}\ \ v_1\ op\ v_2 \longrightarrow v \quad \text{if } v = v_1\,\mathbf{op}\,v_2$$

The rules for primitive operators follow the same principle as those for function application. The first expression is fully evaluated before the evaluation of the second expression starts. When both of the expressions have evaluated to a value, the binary operation denoted by **op** is applied to them to yield the final result. Note that we assume that *op* ranges over primitive operator symbols and that for each such symbol there exists a corresponding primitive operation.

We have presented a particular way of defining operational semantics for the Core which uses small-step transitions and direct substitution. We use the term small-step transition here for the transition of an expression by a single step from one form to another. It is important to note that there are a variety of choices as to how to define operational semantics. For example, in some cases it may be technically more convenient to use big-step transitions or to make use of explicit evaluation environments. The term big-step transition is used for those transitions which involve multiple computational steps. In many cases different choices in formulations do not lead to a change in the meaning of the language and different formulations of semantics can usually be proved to be equivalent. The work by Nielson and Nielson provides useful insights about this topic [NN98].

2.3.4 Type system

Type systems constitute an essential part of our work and we will be presenting a variety of type systems which are designed to capture a variety of phenomena related to mobile computation. We now present a simple monomorphic type system for the Core. The aim of this type system is to show the role of types in classifying values of the language and that type systems adopt a compositional approach to deriving a type for an expression.

Types	τ	::=	$\mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \rightarrow \tau_2$
Type environment	Γ	::=	$[\,] \mid \Gamma[x \mapsto \tau]$

The types of the language consist of the unit type, the type of integers, booleans and function types. The other semantic object of the type system is the type environment Γ which is a finite map from variables to types. The notation $\Gamma[x \mapsto \tau]$ is used to denote the environment Γ extended with the binding of variable x to type τ.

A sentence of the form $\Gamma \vdash e : \tau$ is referred to as a typing judgement. It means that assuming type environment Γ, expression e is well-formed according to the type system and has type τ.

Constant $\quad \Gamma \vdash () : \mathsf{unit} \qquad \Gamma \vdash n : \mathsf{int}$

$\Gamma \vdash \mathsf{true} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{false} : \mathsf{bool}$

The constants are the basic values of the Core and their types are the basic types.

Variable $\quad \Gamma \vdash x : \Gamma(x)$

The type of a variable must be present in the type environment as a binding for the variable.

Function abstraction $\quad \dfrac{\Gamma[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash \mathsf{fn}\ x \Rightarrow e : \tau \rightarrow \tau'}$

The type of a function abstraction is a composition of the type of its formal parameter and the type of its body. It also important to note that the type of the body is derived with respect to an environment where the binding for the formal argument is present. The types on the left and right of the arrow are referred to as the argument type and the result type respectively.

Function application

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\, e_2 : \tau}$$

This rule indicates that for an application to be well-formed according to the type system, the first expression must have a function type and its argument type must match the type of the second expression.

Conditional expression

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1 \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

The first expression of a conditional is required to be of boolean type. The two expressions which constitute the branches can be of any type so long as the types of both expressions are identical. The type of the conditional is the same as the type of its branches.

Local binding

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma[x \mapsto \tau] \vdash e_2 : \tau'}{\Gamma \vdash \mathsf{let} x = e_1\, \mathsf{in} e_2 : \tau'}$$

Two conditions are necessary for the entire let expression to be well-formed. Firstly, the expression of the local declaration must be well-formed. Secondly, it must be possible to derive a type for the body of the expression in the environment extended with the binding obtained from the declaration. The type of the body is also the type of the entire expression.

Primitive operation

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \quad op : (\tau * \tau) \rightarrow \tau'}{\Gamma \vdash e_1\ op\ e_2 : \tau'}$$

We assume that primitive operators have predefined types and that they operate on a pair of values which have identical types. The result type, however may be different from the types of the operands. For example, we regard the test for equality as a primitive operation which tests whether a pair of integers are equal and returns a boolean value as a result.

If we view an expression as a simple program to be executed, the evaluation and typing rules define respectively the dynamic and static phases of its execution. In later chapters, we will exploit extensions of this type system to make static predictions about the dynamic behaviour of programs.

Chapter 3

Estimating Mobile Values

Mobile code languages facilitate the transmission of code between remote sites in a system. A piece of code which is generated at one site can be transmitted for execution at another site which exhibits characteristics different from its place of origin. This poses implementation challenges for mobile code languages. For example, it becomes necessary for compiler writers to consider the heterogeneity of network nodes when generating code. By heterogeneity we mean that different nodes of a network may be of different architectures and therefore support different value representations.

Another implementation challenge involves performance. Transmission of large sizes of code and data may incur significant performance penalties for the system. Minimizing the tranmission overheads becomes an essential goal for implementors.

The ability to statically predict which values may be transmitted to a remote site during the execution of a program can be useful in addressing these implementation challenges. It facilitates code and space optimizations and the development of profiling tools which can be used to tune performance.

It is the aim of this chapter to demonstrate how such predictions can be made by using an approach which is based on type and effect systems. The implementation of the Facile language by Knabe introduced in Chapter 2 provides the framework for our discussion. We define a language which is a subset of the language Facile and investigate in a formal setting the problem of estimating mobile values.

3.1 Application areas

3.1.1 Compiler optimizations

It is argued by Knabe [Kna95] that the first-class nature of functions could prove to be convenient for mobile computation if one could successfully deal with the heterogeneity of network nodes. A common method for dealing with heterogeneity is to adopt a standard transmissible representation. Before transmission, each value is converted into a standard representation determined by its type. This operation is called *marshalling*. Marshalling is a recursive process which decomposes a value until it terminates with the conversion of primitive components such as integers and characters. Upon receipt, the standard representation is converted into the representation appropriate for the receiving machine in an operation called *unmarshalling*. In Facile and most higher-order languages, a function is compiled into a function closure which contains the code of the function together with the bindings from the definition environment of the function. The primitive component of a closure in Facile is the machine code of the function. Converting machine code to a standard representation is a difficult problem and the remaining possibility of translating one native machine code into another one at run-time is mostly impractical.

The fact that functions produced at run-time are just new closures containing old code makes it possible to circumvent this impracticality. One can perform marshalling and generate transmissible representations of code at compile-time and an increased run-time performance can be obtained.

Generating the transmissible representation for each function and storing them, however would be space inefficient. Different approaches to reduce the space cost have been discussed by Knabe [Kna95]. The approach adopted by Facile is to perform compile-time marshalling only for those functions which have been explicitly annotated by the programmer to be potentially transmissible. The identification of potentially transmissible functions automatically without resorting to user annotations has been stated as future work in the dissertation of Knabe [Kna95]. The type system presented in this chapter, which conservatively estimates mobile functions, proposes a solution to this problem. Such an estimation also allows one to infer the locality of functions which are not detected to be mobile by the type system. Knowing that a function is local the compiler can then generate code which is optimized for the local machine.

3.1.2 Cost profiling

Another area where information with respect to mobile entities can be put to use is providing a profile of a program to facilitate reasoning about its transmission overhead.

Many of the mobile computation languages adopt different mechanisms for value transmission and data space management [FPV98, SY97]. Mechanisms of value transmission are classified by Sekiguchi [SY97] as follows. Transmission by *copy* indicates that the value is copied to the data space of the receiver. Values are called *resident* if they always stay at the current place and are never moved to another place. Resident values are referred to by remote references from the outside. Some values are *carried* to the destination and they are referred to by remote references from their previous place. *Proper* values never go out and can never be referred to from the outside. *Take-away* values belong to mobile entities, they go with them and can no longer be referred to from their originating site. *Ubiquitous* values are present in every location so they do not incur any changes in the data space.

In Facile, the transmission of mobile functions occurs by copy whereas local functions are proper values. There are also ubiquitous values. Bindings to these are created dynamically. Data structures which implement channels reside on the node where they were created and they do not move. According to the classification above, Facile channels are resident values; their transmission requires that a network reference be established and maintained.

In this setting, one can expect the cost of computation to be dominated by the cost of maintaining connectivity and copying function closures between remote sites. An estimation of the channels which will be remotely referenced and the functions which will be copied between remote sites would be useful for estimating the overall cost of computation statically. The type system presented in this chapter estimates both mobile functions and mobile channels. It can provide the basis of a cost profiling tool for a Facile-like language.

3.2 Potential mobility

In this section we describe informally what is meant in this chapter by the *potential mobility* of functions and channels. We follow closely the criteria determined by Knabe to guide programmers in identifying potentially mobile functions when using his implementation of Facile. In the following section we will define formally a Facile-like language called Mobile-λ and continue our investigation based on that language. For the purposes of this section, it is sufficient to note that the language has the operators ! and ? which correspond to send and receive actions respectively.

3.2.1 Mobile functions

Any function which is passed directly as an argument to the send operator (!) should be classified as potentially mobile by a static analysis. It is clear that any such function will be transmitted if the execution takes the path on which the send operator occurs. We also know that whenever a function `f` moves, it takes with it the functions contained in its closure. Therefore, we can argue that a function which is referred to by a mobile function and which is not defined within it is also mobile. The functions which are defined within function `f` are mobile only by virtue of being a part of its code. Detecting the mobility of `f` implies the mobility of functions nested within its body.

Example 3.1. The expression below defines two functions `f` and `g` which are local to an expression that sends `g` over a channel `chan`. We assume that `chan` is present in the environment and that function `f` does not refer to any value defined outside its body. Function `f` is a part of the definition environment of function `g` and `g` refers to `f`.

```
let  fun f x = ...;
     fun g x = ...  f x ...
in   chan ! g
```

When `g` is sent to a remote site, `f` will also be sent because it is a part of the closure of `g`. Therefore, we consider `f` to be mobile as well as `g`. If this example was sufficient to illustrate what gives rise to the mobility of functions, all the information needed for automatic detection of mobile functions could be obtained by examining the definition environment of a function which contains the bindings of its free identifiers. However, in the presence of higher-order functions one needs to go beyond the information provided by the definition environment. The following examples illustrate this point.

Example 3.2. Let us consider a function `f` defined as follows.

```
fun f h =  let fun g x = ...  h x ...
           in
             chan ! g
```

It is obvious that the function `g` is mobile. We can also deduce that `h` stands for a function referred to by `g`. However, `h` is neither defined within `f` nor is its binding available in the definition environment of `f`; it is a bound variable. In this situation, all the functions which `h` can be bound to – that is all the functions which `f` can be applied to – are potentially mobile. Detecting these functions is not straightforward since one needs to consider cases such as the following.

Applications of `f`	**Mobile values**
`fun k x = ...  f a ...;`	`a`
`fun k x = ...  f x ...;`	any possible binding of `x`
`fun k x = ...  x a ...;`	
`... k f ...;`	`a`
`fun k h y = ...  h y ...;`	
`... k f ...;`	any possible binding of `y`

Example 3.3. A similar difficulty arises when a higher-order function is transmitted. Let us consider an expression which transmits a function `f` defined as follows.

```
let fun f x =  let fun g y = ...  x ...  y ...
               in g
in chan ! f
```

The function `f` is mobile. The function `g` is defined within `f`. According to our discussion above, there is no immediate reason for `g` to be taken as potentially mobile. However, it should be noted that the function `g` escapes the definition of `f` because it is returned as a result. It may be the case that `g` is subsequently transmitted by some other code which receives `f`. We consider functions such as `g` and the functions which may be transmitted by `g` also as potentially mobile.

3.2.2 Mobile channels

The arguments for mobile functions above also apply to mobile channels and our criteria for identifying mobile channels is the same as for functions. This is easily justified as channels are also first-class values and all first-class values are treated uniformly in Facile-like languages.

3.2.3 Related work

As is demonstrated by the preceding examples, in languages with higher-order functions the flow of control from one program point to another is not easily detectable. This is because a function can be passed around and subsequently called from multiple sites in the program. A wide range of analyses have been devised to approximate which functions can be called from a particular point in the program. Some instances of these static analyses are known as closure analysis, set-based analysis and constraint-based analysis which differ in their formulations, the precision they offer and their practicality [Shi91, GFH97, NN97, JW95, Ste96, Hei94b].

More recently, some authors have pointed out the intuitive connection between reasoning about types and control flow in higher-order languages in the sense that they both derive invariants about the potential bindings of variables in a program. Research has been carried out in extending control flow systems to perform type analyses [PS95] and in extending type systems to perform control flow analyses [TJ92].

It has also been observed that type inference and control flow analysis are based on different perspectives. Type inference systems reason locally by associating a type with each expression and deriving a type for the program compositionally whereas control flow analyses reason globally and are usually not compositional. Another direction of research has focused on systematic comparisons of type systems and control flow systems by establishing correspondences between certain type systems and control flow analyses [Hei94a, PO95].

Variations for control flow analyses developed for concurrent programming languages such as [GFH97, NN97, Ste96] could offer a solution to the problem of detecting potentially mobile values. However, an effect-based analysis which exploits the existing type system would be more easily applicable. The latter is also the one which is proposed by [Kna95]. We design an annotated type and effect system which exposes the overall communication behaviour of a program. The information captured by the effects are then analysed to estimate the mobile values.

The values which are not detected to be mobile by our type system are guaranteed to be used only locally. This relates our work to the works on locality inference by using type systems [Mor99, Sew98]. The languages considered in these works are different to the one we consider in this chapter. The exploitation of an effect system for locality inference is another point which distinguishes our type system from theirs.

3.3 Mobile-λ

In this section we introduce the language Mobile-λ which extends the Core language of Chapter 2 with primitives for communication between remote sites. The computational model induced by Mobile-λ provides a sufficient level of generality to model the transmission of functions and channels in a Facile-like language which is the main interest of this chapter.

3.3.1 Abstract syntax

The abstract syntax for the sequential core of Mobile-λ is similar to that of the Core language from Chapter 2. The only difference is that function abstractions are annotated with labels. The purpose of labels (l) is to uniquely identify functions. Mobile-λ extends the Core with constructs to express dynamic channel allocation and sending and receiving values over channels. Channel allocation expressions are also annotated with labels which serve a similar purpose to those of function labels; they uniquely identify channel allocation points in a program. Note that all of the labels in a program are required to be distinct in order to serve as unique identifications. These labels would typically be internal to a compiler of Mobile-λ rather than being explicitly provided by authors of code.

Labels	l	::=	$l_1 \mid l_2 \mid \ldots$	
Expressions	e	::=	$\ldots$	expressions of the Core
			$\mid \mathsf{fn}^l\, x \Rightarrow e$	function abstraction
			$\mid \mathsf{chan}^l()$	channel allocation
			$\mid e_1!e_2$	send
			$\mid e?$	receive

Figure 3.1: Abstract Syntax for Mobile-λ

Convention Throughout the thesis, we present the abstract syntax of a number of languages as we have done for Mobile-λ above. To avoid repetition the figures do not include syntactic forms which are identical to those of the Core Language (see Figure 2.6).

We refer to the *free identifiers* of an expression e as $FI(e)$. Function abstractions and let bindings are the only forms of expressions which bind identifiers. An abstraction of the form $\mathsf{fn}^l\, x \Rightarrow e$ binds x in e and a let expression of the form $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ binds x in e_2. Formally, $FI(\mathsf{fn}^l\, x \Rightarrow e) = FI(e) \setminus \{x\}$ and $FI(\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2) = FI(e_1) \cup (FI(e_2) \setminus \{x\})$. The definition of FI extends to other forms of expressions in the obvious way; the set of the free variables of an expression is the union of the free variables of its subexpressions.

3.3.2 Dynamic semantics

Our model of a system is a collection of named sites each of which hosts the execution of a single Mobile-λ expression. The names of the sites, ranged over by s, is drawn from a finite set with elements $s_1 \ldots s_n$.

3.3.2.1 Environments and values

The definition of the dynamic semantics makes use of environments which are defined as finite maps from identifiers to values. We write $Dom(E)$ for the domain of an environment E. The explicit use of environments in the definition of the dynamic semantics allows us to capture directly the mobility of values present in the definition environment of a function.

Evaluation environments	E	$::=$	$[\,] \mid E[x \mapsto v]$	
Values	v	$::=$	c	constants
			$\mid k$	channel identifers
			$\mid \langle l, E, x, e \rangle$	function closures

Values consist of basic constants, channel identifiers and function closures. The closure of a function encapsulates the label l of the function, an environment E, the formal parameter x and the function body e. The role of the environment in the closure is to provide the bindings for the free identifiers of the function body.

Definition 3.1 (Ubiquitous identifiers). Let E_s denote the evaluation environment at site s. We define the set U as the set of identifiers which are present in the domain of each evaluation environment in the system. Let $S = \{s \mid s \text{ is a site in the system}\}$.

$$U = \{x \mid x \in Dom(E_s) \text{ for all } s \in S\}.$$

Definition 3.2 (Environment composition). Two disjoint environments E and E' can be composed by the operation $\oplus$ such that

$$(E \oplus E')(x) = \begin{cases} E(x) & \text{if } x \in Dom(E) \text{ and } x \notin Dom(E') \\ E'(x) & \text{if } x \in Dom(E') \text{ and } x \notin Dom(E) \end{cases}$$

Each environment E in a system can be understood as a composition of two disjoint parts, that is $E = E_u \oplus E_{local}$, where E_u binds the ubiquitous identifiers U and E_{local} binds the identifers which are local to the site in question.

Definition 3.3 (Environment narrowing). Given an evaluation environment E and a set of identifiers V, $E \downharpoonleft V$ represents the evaluation environment which is obtained from E by removing the bindings of the identifiers not present in V or present in U.

$$\begin{aligned} E \downharpoonleft V = E' \quad & \text{where } Dom(E') = V \setminus U \\ & \text{and } E'(x) = E(x) \text{ for all } x \in Dom(E'). \end{aligned}$$

We define the dynamic semantics of Mobile-λ by using small-step transitions between system states. Since we use explicit environments rather than direct substitutions, some transitions give rise to forms of expressions which do not conform to the abstract syntax presented in Figure 5.1. In order to circumvent this problem we introduce intermediate forms of expressions (*ie*) as in [NNH99a]. Intermediate expressions extend the expressions of the abstract syntax with function closures and bind expressions. Without the inclusion of function closures in the intermediate expressions we would not be able to express the evaluation of a function abstraction. The need for bind expressions will become more clear when we present the evaluation rules below. At this moment it suffices to say that a bind expression of the form bind E in ie represents an expression where the bindings of the free identifiers of ie are recorded in E. The environments of nested bind expressions are likened to frames of a run-time stack in [NNH99a].

$$\begin{array}{lrcl} \textbf{Intermediate expressions} & ie & ::= & c \mid x \mid \mathsf{fn}^l\, x \Rightarrow e \mid ie_1\; ie_2 \\ & & & \mid ie_1\; op\; ie_2 \mid \mathsf{if}\; ie_1\; \mathsf{then}\; e_2\; \mathsf{else}\; e_3 \\ & & & \mid \mathsf{let}\; x = ie_1\; \mathsf{in}\; e_2 \\ & & & \mid \mathsf{chan}^l() \mid ie_1!ie_2 \mid ie? \\ & & & \mid k \mid \langle l, E, x, e \rangle \mid \mathsf{bind}\; E\; \mathsf{in}\; ie \end{array}$$

3.3.2.2 Evaluation rules

A system state is represented by a channel identifier set CI and a process pool P in the style of [BMT92]. The channel identifier set contains the channel identifiers created so far in the computation and the process pool is a set of tuples which comprise a site name, an evaluation environment and an expression. An element of the process pool, written as $[(s, E) : ie]$, indicates that expression ie is to be executed at site

s with respect to environment E. We use the notation $P[(s,E):ie]$ to denote the process pool $P \cup [(s,E):ie]$. A single step transition between system states is written as $CI,P \xrightarrow{a} CI',P'$. The annotation a on the arrow represents the observable actions.

Actions	a	::=	ε	no communication
			$\mid s[\mathsf{new}\,k]$	channel allocation
			$\mid (s_1[k!v], s_2[k?v])$	communication of a value

The actions which we would like to observe are channel allocation at a site and the transmission of a value between two remote parties on a shared channel.

The evaluation rules are presented in three parts in Figures 3.2, 3.3 and 3.4. The rules for the sequential subset correspond to the standard evaluation rules for call-by-value functional languages put into a distributed context.

Rules of Figure 3.2 The rule (id) is applied in the evaluation of value identifiers. The value of an identifier is obtained by looking it up in the environment. The rule (fn) shows how a function abstraction evaluates to a closure. The definition environment of the function is narrowed down according to Definition 3.3 before it is included in the closure. Rules (app-1) and (app-2) are similar to those of the Core Language from Chapter 2. The rule (app-3) gives rise to a bind expression. The environment part of the closure is extended with the binding of the argument. It is then enclosed in a bind expression with the body of the function. The domain of environment $E'[x \mapsto v]$ contains the local identifiers of e and it can be discarded when the evaluation of e is complete. However, the environment E may contain the bindings of identifiers which are necessary for the rest of the evaluation. If we had $[(s,E'):e]$ in the rule instead of $[(s,E):\mathsf{bind}\,E'[x \mapsto v]\,\mathsf{in}\,e]$ we would override E with $E'[x \mapsto v]$ and not be able to recover it again. The rule (bind-1) shows that an expression within a bind construct is evaluated with respect to the composition of the ubiquitous environment and the environment which is enclosed in the bind construct. When the expression has been fully evaluated the environment can be discarded as shown in rule (bind-2).

(id) $CI, P[(s,E) : x] \xrightarrow{\varepsilon} CI, P[(s,E) : E(x)]$

(fn) $CI, P[(s,E) : \mathsf{fn}^l\, x \Rightarrow e] \xrightarrow{\varepsilon} CI, P[(s,E) : \langle l, E', x, e\rangle]$
where $E' = E \downharpoonright FI(\mathsf{fn}^l\, x \Rightarrow e)$

(app-1) $$\frac{CI, P[(s,E) : ie_1] \xrightarrow{a} CI', P'[(s,E) : ie_1']}{CI, P[(s,E) : ie_1\ ie_2] \xrightarrow{a} CI', P'[(s,E) : ie_1'\ ie_2]}$$

(app-2) $$\frac{CI, P[(s,E) : ie_2] \xrightarrow{a} CI', P'[(s,E) : ie_2']}{CI, P[(s,E) : v\ ie_2] \xrightarrow{a} CI', P'[(s,E) : v\ ie_2']}$$

(app-3) $CI, P[(s,E) : \langle l, E', x, e\rangle\ v] \xrightarrow{\varepsilon} CI, P[(s,E) : \mathsf{bind}\, E'[x \mapsto v]\, \mathsf{in}\, e]$

(bind-1) $$\frac{CI, P[(s, E_u \oplus E') : ie] \xrightarrow{a} CI', P'[(s, E_u \oplus E') : ie']}{CI, P[(s,E) : \mathsf{bind}\, E'\, \mathsf{in}\, ie] \xrightarrow{a} CI', P'[(s,E) : \mathsf{bind}\, E'\, \mathsf{in}\, ie']}$$

(bind-2) $CI, P[(s,E) : \mathsf{bind}\, E'\, \mathsf{in}\, v] \xrightarrow{\varepsilon} CI, P[(s,E) : v]$

Figure 3.2: Evaluation Rules (Part 1)

Rules of Figure 3.3 We refer the reader to Chapter 2 for the explanations of these rules. The only rule which might be unfamiliar is (let-2) which becomes applicable when the evaluation of the first expression has been completed. The environment is then narrowed down according to Definition 3.3 before being enclosed in a bind expression along with the body of the let expression.

Rules of Figure 3.4 The rule (chan) for channel allocation states that a fresh channel identifier is generated upon the execution of a channel allocation expression. We assume that a channel identifier k is represented by a tuple which includes the label l of the corresponding channel allocation expression, the identifier of the site s it is created

(if-1) $$\frac{CI,P[(s,E):ie_1] \xrightarrow{a} CI',P'[(s,E):ie'_1]}{CI,P[(s,E):\mathsf{if}\ ie_1 \mathsf{then}\ e_2\ \mathsf{else}\ e_3] \xrightarrow{a} CI',P'[(s,E):\mathsf{if}\ ie'_1 \mathsf{then}\ e_2\ \mathsf{else}\ e_3]}$$

(if-2) $$CI,P[(s,E):\mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3] \xrightarrow{\varepsilon} CI,P[(s,E):e_2]$$

(if-3) $$CI,P[(s,E):\mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3] \xrightarrow{\varepsilon} CI,P[(s,E):e_3]$$

(let-1) $$\frac{CI,P[(s,E):ie_1] \xrightarrow{a} CI',P'[(s,E):ie'_1]}{CI,P[(s,E):\mathsf{let}\ x = ie_1\ \mathsf{in}\ e_2] \xrightarrow{a} CI',P'[(s,E):\mathsf{let}\ x = ie'_1\ \mathsf{in}\ e_2]}$$

(let-2) $$CI,P[(s,E):\mathsf{let}\ x = v\ \mathsf{in}\ e] \xrightarrow{\varepsilon} CI,P[(s,E):\mathsf{bind}\ E'[x \mapsto v]\ \mathsf{in}\ e]$$
where $E' = E \downharpoonright FI(e)$

(op-1) $$\frac{CI,P[(s,E):ie_1] \xrightarrow{a} CI',P'[(s,E):ie'_1]}{CI,P[(s,E):ie_1\, op\, ie_2] \xrightarrow{a} CI',P'[(s,E):ie'_1\ op\ ie_2]}$$

(op-2) $$\frac{CI,P[(s,E):ie_2] \xrightarrow{a} CI',P'[(s,E):ie'_2]}{CI,P[(s,E):v\ op\ ie_2] \xrightarrow{a} CI',P'[(s,E):v\ op\ ie'_2]}$$

(op-3) $$CI,P[(s,E):v_1\ op\ v_2] \xrightarrow{\varepsilon} CI',P'[(s,E):v] \text{ where } v = v_1\ \mathbf{op}\ v_2$$

Figure 3.3: Evaluation Rules (Part 2)

at and an integer i which is freshly generated each time a new channel is allocated at that site ($k = (l,s,i)$). The rule (com) for communication illustrates that in order for a value to be transmitted from one site to another, the communicating expressions need to synchronize. Unless the synchronization takes place none of the parties can resume their computation.

Our choice of style in defining the dynamic semantics of Mobile-λ has been influenced by the need to provide a suitable framework for conducting the proofs of some

(chan) $$CI, P[(s,E) : \mathsf{chan}^l()] \xrightarrow{a} CI \cup k, P[(s,E) : k]$$
where $a = s[\mathsf{new}\, k]$ and $k \notin CI$

(send-1) $$\frac{CI, P[(s,E) : e_1] \xrightarrow{a} CI', P'[(s,E) : e_1']}{CI, P[(s,E) : e_1 \,!\, e_2] \xrightarrow{a} CI', P'[(s,E) : e_1' \,!\, e_2]}$$

(send-2) $$\frac{CI, P[(s,E) : e_2] \xrightarrow{a} CI', P'[(s,E) : e_2']}{CI, P[(s,E) : k \,!\, e_2] \xrightarrow{a} CI', P'[(s,E) : k \,!\, e_2']}$$

(receive) $$\frac{CI, P[(s,E) : e_1] \xrightarrow{a} CI', P'[(s,E) : e_1']}{CI, P[(s,E) : e_1 ?] \xrightarrow{a} CI', P'[(s,E) : e_1' ?]}$$

(com) $$CI, P[(s_1,E_1) : k \,!\, v][(s_2,E_2) : k?] \xrightarrow{a} CI, P[(s_1,E_1) : ()][(s_2,E_2) : v]$$
where $a = (s_1[k!v], s_2[k?v])$

Figure 3.4: Evaluation Rules (Part 3)

properties enjoyed by our type system. For example, we include labels in the representation of channels and functions in the dynamic semantics. This is because our type system makes use of labels to trace the identities of functions and channels statically. Labels of the dynamic semantics prove to be useful in establishing a correspondence between the objects of the static and the dynamic semantics.

Definition 3.4 (Labels). The function *Labels* is defined on the values of the dynamic semantics (see Section 3.3.2.1 for the definition of values). It collects the labels of functions and channels which are of interest in detecting potential mobility. In the case of function closures the labels of values in the environment part of the closure are collected as well as the label of the closure itself.

$$Labels(v) = \begin{cases} \emptyset & \text{if } v = c \\ \{l\} & \text{if } v = k = (l,s,i) \\ \{l\} \cup \bigcup \{Labels(E(x)) \mid x \in Dom(E)\} & \text{if } v = \langle l,E,x,e \rangle \end{cases}$$

3.4 Type system

In this section we design a polymorphic type and effect system for Mobile-λ. Our aim is to estimate the functions and channels that a Mobile-λ expression can possibly transmit by analysing the types and effects associated with it. We make use of the labels provided in the syntax to trace the flow of values through the computation. The essential idea is to incorporate these labels into the types and effects so that we can eventually extract those which are of interest to us.

3.4.1 Semantic objects

Our type system views types (τ) as a pair consisting of two components: a raw type ($\bar{\tau}$) and a mobility annotation (μ). Raw types classify values in the conventional sense whereas mobility annotations estimate the identities of values which become mobile upon the transmission of the value of that raw type. We also have communication effects (κ) as annotations on function types. These stand for the communication actions which may be triggered by a function when it is applied.

Mobility annotations	μ	::=	$\emptyset \mid \{l\} \mid \gamma \mid \mu_1 \cup \mu_2$
Raw types	$\bar{\tau}$	::=	$\mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{chan}[\tau] \mid \tau_1 \xrightarrow{\kappa} \tau_2 \mid \alpha$
Types	τ	::=	$(\bar{\tau}, \mu)$
Effects	κ	::=	$\emptyset \mid \{\mathtt{new}\,\mu\ \mathsf{for}\ \tau\} \mid \{\mathsf{send}\,\tau\,\mathsf{on}\,\mu\} \mid \{\mathsf{recv}\,\tau\,\mathsf{on}\,\mu\}$
			$\mid \beta \mid \kappa_1 \cup \kappa_2$

Mobility annotations are represented by sets. We associate the empty set with base types as we are not concerned with the mobility of values of base types. For example, if the raw type indicates that the expression is of type int, this information is sufficient. We need not estimate which particular integers its value can be.

However, as we are interested in the mobility of channels and functions we associate them with non-empty annotations. It is not always possible to statically infer precise information about the value of an expression. To facilitate estimations we allow mobility annotations to be a union of simpler annotations. The meta-variable γ stands for mobility annotation variables. Mobility annotation variables prove to be useful in typing functions which require functions or channels as arguments. One can write well-typed functions which behave uniformly over a set of arguments which may differ in their mobility annotations.

The set κ estimates the communication effects which may occur during the evaluation of an expression. An expression is assigned the effect $\emptyset$ if its evaluation incurs no communication. The effects $\{\mathtt{new}\mu\ \mathsf{for}\ \tau\}$,$\{\mathsf{send}\,\tau\,\mathsf{on}\,\mu\}$ and $\{\mathsf{recv}\,\tau\,\mathsf{on}\,\mu\}$ are assigned if its evaluation may incur the allocation of a channel for carrying a specified type of values, send and receive actions respectively. The effects can be merged by set union as was the case for mobility annotations. The meta-variable β stands for effect variables.

3.4.1.1 Variables and substitutions

We have included variables α, β, γ as static semantic objects to be able to express polymorphism of functions. We write $FV(\tau)$ for the set of free raw type, mobility annotation and effect variables in τ. Vector notation is used to represent sequences of variables, for example $\vec{\alpha}, \vec{\beta}, \vec{\gamma}$. Finite maps which map raw type variables to raw types, effect variables to effects and mobility annotation variables to mobility annotations are called *substitutions*.

Mobility annotations:

$FV(\emptyset) = \emptyset$

$FV(l) = \emptyset$

$FV(\gamma) = \{\gamma\}$

$FV(\mu_1 \cup \mu_2) = FV(\mu_1) \cup FV(\mu_2)$

Raw types:

$FV(\mathsf{unit}) = \emptyset \quad FV(\mathsf{int}) = \emptyset \quad FV(\mathsf{bool}) = \emptyset$

$FV(\mathsf{chan}[\tau]) = FV(\tau)$

$FV(\tau \xrightarrow{\kappa} \tau') = FV(\tau) \cup FV(\tau') \cup FV(\kappa)$

$FV(\alpha) = \{\alpha\}$

Effects:

$FV(\emptyset) = \emptyset$

$FV(\{\mathtt{new}\mu\ \mathsf{for}\ \tau\}) = FV(\tau) \cup FV(\mu)$

$FV(\{\mathsf{send}\,\tau\,\mathsf{on}\,\mu\}) = FV(\tau) \cup FV(\mu)$

$FV(\{\mathsf{recv}\,\tau\,\mathsf{on}\,\mu\}) = FV(\tau) \cup FV(\mu)$

$FV(\beta) = \{\beta\}$

$FV(\kappa' \cup \kappa') = FV(\kappa) \cup FV(\kappa')$

Types:

$FV(\bar{\tau}, \mu) = FV(\bar{\tau}) \cup FV(\mu)$

Type schemes and environments are defined as in polymorphic type systems for languages with a functional core such as Standard ML. Type schemes (σ) are obtained by universally quantifying types over a set of raw type, effect and mobility annotation variables.

Type schemes	σ	$::=$	$\forall\vec{\alpha}\vec{\beta}\vec{\gamma}.\tau$
Type environments	Γ	$::=$	$[\,] \mid \Gamma[x \mapsto \sigma]$

The context in which an expression is associated with a raw type, an effect and a mobility annotation is represented by a static environment Γ, which maps identifiers to type schemes. The notation $\Gamma[x \mapsto \sigma]$ is used for adding element x to the environment Γ, overriding the existing binding if x is already in the domain of Γ. We refer to the domain of an environment Γ as $Dom(\Gamma)$.

The definition of FV above extends to type schemes by $FV(\forall\vec{\delta}.\tau) = FV(\tau) \setminus \vec{\delta}$. It also extends pointwise to type environments. Substitutions on type schemes are defined as in [TJ94]. Let $\theta, \theta', \theta''$ be three substitutions. $\theta(\forall\vec{\delta}\tau) = \forall\vec{\delta}'.\theta''\theta'(\tau)$ where $\theta' = \{\vec{\delta} \mapsto \vec{\delta}'\}$ is a renaming of the bound variables by fresh variables $\vec{\delta}'$ which are not free in τ or $\theta\tau$. The substitution $\theta''\delta$ is defined as δ if $\delta \in \vec{\delta}'$ and $\theta\delta$ otherwise.

Definition 3.5 (Type generalization). A type scheme $\sigma = \forall\vec{\alpha}\vec{\beta}\vec{\gamma}.\tau$ generalizes a type τ', written as $\sigma \succ \tau'$, if there exists a substitution θ over the bound variables of τ such that $\theta\tau = \tau'$. Types are generalized to type schemes by the operation *Gen*. A variable cannot be generalized if it is free in the environment Γ or the effect κ.

$$\begin{aligned} Gen(\Gamma,\kappa,\tau) = \quad & \text{let } \{\vec{\alpha},\vec{\beta},\vec{\gamma}\} = FV(\tau) \setminus (FV(\Gamma) \cup FV(\kappa)) \\ & \text{in } \forall\vec{\alpha}\vec{\beta}\vec{\gamma}.\,\tau. \end{aligned}$$

The condition which prevents the generalization of variables which are free in the type environment is standard from the Damas-Milner polymorphic type discipline for purely functional languages [DM82]. The additional condition which requires them not to be observable in the effect is imposed to ensure that the presence of primitives for communication do not compromise type safety. We follow closely the work on polymorphic types and effects for the Facile language which proposes a generalization criterion similar to ours [Tho94].

Definition 3.6 (TypeOf). The types of the basic constants unit, integers and booleans do not depend on the typing context and are defined as follows:

$$\begin{array}{ll} TypeOf(()) = (\mathsf{unit}, \emptyset) & TypeOf(n) = (\mathsf{int}, \emptyset) \\ TypeOf(\mathsf{true}) = (\mathsf{bool}, \emptyset) & TypeOf(\mathsf{false}) = (\mathsf{bool}, \emptyset) \end{array}$$

We define now define an operation which is employed in the type system to collect the labels of interest in the process of deriving a type for an expression.

Definition 3.7 (Extracting annotations). Suppose that U is a predefined set which contains the identifiers of the ubiquitous values. Given a type environment and a set of identifiers, the operation M extracts the mobility annotations of the identifiers from their types.

$$M(\Gamma, \{x\}) = \begin{cases} \emptyset & \text{if } x \in U \text{ or } x \notin Dom(\Gamma) \\ \mu & \text{otherwise where } \Gamma(x) = (\bar{\tau}, \mu) \end{cases}$$

$$M(\Gamma, \{x\} \cup X) = M(\Gamma, \{x\}) \cup M(\Gamma, X)$$

The labels of identifiers of ubiquitous values are discarded. This is because these values are present at every site and hence are of no interest in terms of estimating mobility.

3.4.2 Typing rules

The static semantics for the language assigns a type and an effect to each expression. This is represented by a judgement of the form $\Gamma \vdash e : \tau, \kappa$. The typing rules are given in Figures 3.5 and 3.6. We comment on some of the rules below pointing out the characteristic features of our type system.

The typing rule (id) shows that an identifier can be assigned any type which is an instance of the type scheme which it is bound to in the type environment.

The typing rule (fn) for functions is essential for our type system. It associates a non-trivial mobility annotation with a function. This annotation serves as an estimation of the labels of those values which move with the function. The operation M of Definition 3.7 inspects the typing environment of the function to collect the labels of functions and channels which are referred to by the function body and which are not ubiquitous. These labels are merged with the label of the function itself and recorded as its mobility annotation.

(con) $\Gamma \vdash c : TypeOf(c), \emptyset$

(id) $$\frac{\Gamma(x) = \sigma \quad \sigma \succ \tau}{\Gamma \vdash x : \tau, \emptyset}$$

(fn) $$\frac{\Gamma[x \mapsto \tau] \vdash e : \tau', \kappa \quad \mu = M(\Gamma, FI(\mathsf{fn}^l\, x \Rightarrow e))}{\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e : (\tau \xrightarrow{\kappa} \tau', \{l\} \cup \mu), \emptyset}$$

(app) $$\frac{\Gamma \vdash e_1 : (\tau \xrightarrow{\kappa} \tau', \mu), \kappa' \quad \Gamma \vdash e_2 : \tau, \kappa''}{\Gamma \vdash e_1\, e_2 : \tau', \kappa \cup \kappa' \cup \kappa''}$$

(if) $$\frac{\Gamma \vdash e_1 : (\mathsf{bool}, \emptyset) \quad \Gamma \vdash e_2 : (\bar{\tau}, \mu), \kappa \quad \Gamma \vdash e_3 : (\bar{\tau}, \mu'), \kappa'}{\Gamma \vdash \mathsf{if}\ e_1 \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\bar{\tau}, \mu \cup \mu'), \kappa \cup \kappa'}$$

(let) $$\frac{\Gamma \vdash e_1 : \tau, \kappa \quad \Gamma[x \mapsto Gen(\Gamma, \kappa, \tau)] \vdash e_2 : \tau', \kappa'}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau', \kappa \cup \kappa'}$$

(op) $$\frac{\Gamma \vdash e_1 : (\bar{\tau}, \emptyset), \kappa \quad \Gamma \vdash e_2 : (\bar{\tau}, \emptyset), \kappa' \quad op : (\bar{\tau} * \bar{\tau}) \rightarrow \bar{\tau}'}{\Gamma \vdash e_1\ op\ e_2 : (\bar{\tau}', \emptyset), \kappa \cup \kappa'}$$

(subs) $$\frac{\Gamma \vdash e : \tau, \kappa \quad \kappa \subseteq \kappa'}{\Gamma \vdash e : \tau, \kappa'}$$

Figure 3.5: Typing Rules for the Core

The typing of the conditional expressions by rule (if) requires the raw types of the branches to be identical. Their mobility annotations are merged by set union to obtain the mobility annotation of the whole expression.

Regarding polymorphism, we adopt the standard discipline employed in languages with a functional core and allow type generalization to be performed in the (let) rule only.

(chan)	$\Gamma \vdash \mathsf{chan}^l() : (\mathsf{chan}[\tau], l), \{\mathtt{new}\, l \text{ for } \tau\}$
(send)	$\dfrac{\Gamma \vdash e_1 : (\mathsf{chan}[\tau], \mu), \kappa \quad \Gamma \vdash e_2 : \tau, \kappa'}{\Gamma \vdash e_1 ! e_2 : \mathsf{unit}, \kappa \cup \kappa' \cup \{\mathsf{send}\, \tau\, \mathsf{on}\, \mu\}}$
(receive)	$\dfrac{\Gamma \vdash e : (\mathsf{chan}[\tau], \mu), \kappa}{\Gamma \vdash e? : \tau, \kappa \cup \{\mathsf{recv}\, \tau\, \mathsf{on}\, \mu\}}$

Figure 3.6: Typing Rules for Extensions

The subsumption rule (subs) allows an expression to admit a larger communication effect than seems to be necessary. We adopt the approach which is referred to as early subsumption in [NN94] and subeffecting in [TJ92].

The rule (chan) assigns the mobility annotation l to a channel allocation expression labelled with l. The type of a channel should be in accordance with the values it communicates. In a typing derivation this information has to be derived from the context of the allocation expression. By including τ in the effect of the expression, we make sure that in an attempt to generalize the type of the channel, the type of the values it communicates will be taken into consideration.

The rule (send) always assigns the type unit to a send expression whereas the effect embodies the type of the transmitted value. The fact that a send effect is parameterized over a mobility annotated type makes it possible to analyse these effects further to expose potentially mobile values. We address this in Section 3.6.

The rule (receive) is applied in typing a receive expression. As a receive expression evaluates to the value it receives, its type is expected to be the same as that of the received value.

3.4.3 Examples

Having defined Mobile-λ and its type system we now present some examples which are inspired by the examples from Section 3.2.

Example 1:
$$
\begin{aligned}
e_{let} = \quad & \mathsf{let} \quad \mathsf{f} = \mathsf{fn}^{l_f}\,\mathsf{x} \Rightarrow e_f \\
& \mathsf{in} \quad \mathsf{let}\ \mathsf{g} = \mathsf{fn}^{l_g}\,\mathsf{x} \Rightarrow e_g \\
& \qquad\ \ \mathsf{in}\ \mathsf{ch}\ !\ \mathsf{g}
\end{aligned}
$$

Let us assume the following about f and g:

- f and g are functions from integers to integers with no observable effect;
- e_f contains no free identifiers; and
- f is the only free identifier in e_g.

Then the following is a possible type annotated version of the expression e_{let}. We assume that the expression is typed with respect to an initial environment Γ where $\Gamma = [\mathsf{ch} \mapsto (\mathsf{chan}[\tau_g], \{l_c\})]$ and $\tau_g$is the type of function g shown below.

$$
\begin{aligned}
e_{let} = \quad & \mathsf{let} \quad \mathsf{f} = \mathsf{fn}^{l_f}\,\mathsf{x} \Rightarrow e_f : ((\mathsf{int},\emptyset)\xrightarrow{\emptyset}(\mathsf{int},\emptyset), \{l_f\}), \emptyset \\
& \mathsf{in} \quad \mathsf{let}\ \mathsf{g} = \mathsf{fn}^{l_g}\,\mathsf{x} \Rightarrow e_g : ((\mathsf{int},\emptyset)\xrightarrow{\emptyset}(\mathsf{int},\emptyset), \{l_g, l_f\}), \emptyset \\
& \qquad\ \ \mathsf{in}\ \mathsf{ch}\ !\ \mathsf{g} : (\mathsf{unit}, \emptyset), \{\mathsf{send}\,\tau_g\,\mathsf{on}\,\{l_c\}\}
\end{aligned}
$$

The effect $\{\mathsf{send}\,\tau_g\,\mathsf{on}\,\{l_c\}\}$ of the expression e_{let} conveys the information that when a value of type τ_g is transmitted on the channel with mobility annotation $\{l_c\}$, the functions whose labels are embodied in the type τ_g will be mobile. The type τ_g is $((\mathsf{int},\emptyset)\xrightarrow{\emptyset}(\mathsf{int},\emptyset), \{l_g, l_f\})$. Therefore, we know that functions with labels l_g and l_f will become mobile when the expression e_{let} is evaluated. In Section 3.6 we present an analysis which shows how to extract the mobility annotations from the types systematically, such as extracting l_g and l_f from the type τ_g in this example.

Example 2:
$$
\begin{aligned}
& \mathsf{let}\ \mathsf{f} = \mathsf{fn}^{l_f}\mathsf{h}\ \Rightarrow\ \mathsf{fn}^{l_{f1}}\mathsf{ch}\ \Rightarrow\ \mathsf{let}\ \mathsf{g} = \mathsf{fn}^{l_g}\mathsf{x}\ \Rightarrow\ \mathsf{h}\,\mathsf{x} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \ \mathsf{ch}\,!\,\mathsf{g} \\
& \mathsf{in}\ \ (**\ \ \mathit{scope\ \ of}\ \ \mathsf{f}\ \ **)
\end{aligned}
$$

In this example function f is a higher-order function which takes a function and a channel as arguments. Its body consists of a let expression. We assume that Γ, which is given below, is the type environment in which the body of function f is typed.

- $\Gamma = [\mathsf{h} \mapsto (\bar{\tau}, \gamma_1), \mathsf{ch} \mapsto (\mathsf{chan}[(\bar{\tau}, \{l_g\} \cup \gamma_1)], \gamma_2)]$
  where $\bar{\tau} = (\mathsf{int}, \emptyset) \xrightarrow{\beta} (\mathsf{int}, \emptyset)$.

The typing environment suggests that the actual argument to which h is bound must be a function from integers to integers but nothing specific is assumed about its observable effect. The actual argument to which ch is bound must be a channel which carries functions from integers to integers. We additionally know that the function with label l_g and the function to which h is bound are carried on this channel. The following is a possible type-annotated version of the let expression. Note that for readability we use the abbreviations τ_h for $\Gamma(\mathsf{h})$, τ_{ch} for $\Gamma(\mathsf{ch})$, and κ_f for $\{\mathsf{send}\,((\mathsf{int}, \emptyset) \xrightarrow{\beta} (\mathsf{int}, \emptyset), \{l_g\} \cup \gamma_1)\,\mathsf{on}\,\gamma_2\}$.

$$
\begin{array}{ll}
\mathsf{let} & \mathsf{f} = \mathsf{fn}^{l_f}\mathsf{h} \;\Rightarrow\; \mathsf{fn}^{l_{f1}}\mathsf{ch} \;\Rightarrow \\
& \quad \mathsf{let}\ \mathsf{g} = \mathsf{fn}^{l_g}x \;\Rightarrow\; \mathsf{h}\,\mathsf{x} : (\bar{\tau}, \{l_g\} \cup \gamma_1), \emptyset \\
& \quad\ \ \mathsf{in}\ \ \mathsf{ch}\,!\,\mathsf{g} : \forall \gamma_1 \gamma_2 \beta.(\tau_h \xrightarrow{\emptyset} (\tau_{ch} \xrightarrow{\kappa_f} (\mathsf{unit}, \emptyset)), \{l_{f1}\}), \{l_f\}), \emptyset \\
\mathsf{in} & (**\ \ \mathit{scope\ \ of}\ \ \mathsf{f}\ \ **)
\end{array}
$$

Since the type of τ_h and τ_{ch} do not appear free in the environment in which f is typed, the operation *Gen* from Definition 3.5 would generalize γ_1,γ_2 and β. This means that the function f can be applied uniformly to any function from integers to integers.

We could now convince the reader that the type of f includes all the information we seek. Whenever f is fully applied, its polymorphic type will be specialized according to those of its arguments. The generalized mobility annotations γ_1 of h and γ_2 of ch will be specialized to the mobility annotation of its function argument and its channel argument respectively. For example, if at some point in the scope of f, there appears an expression f k ch1 where k is a function with the mobility annotation $\{l_k\}$ and the observable effect κ_k and ch1 is a channel of the correct type with the mobility annotation $\{l_{ch1}\}$ then the overall effect associated with this application will include $\{\mathsf{send}\,((\mathsf{int}, \emptyset) \xrightarrow{\kappa_k} (\mathsf{int}, \emptyset), \{l_g, l_k\})\,\mathsf{on}\,\{l_{ch1}\}\}$.

3.5 Formal properties of the type system

In this section we prove the soundness of the type system defined in Section 3.4 with respect to the dynamic semantics presented in Section 3.3. This involves proving that types are preserved under transitions of the system, characterizing the run-time errors for Mobile-λ programs and showing that the evaluation of well-typed programs do not give rise to run-time errors.

Mobile-λ programs can allocate channels dynamically. This implies that the intermediate expressions which describe the intermediate steps of evaluation can include channel identifiers which cannot be known statically. Therefore, the static environment Γ alone would not be sufficient to keep track of the types of intermediate expressions. In order to be able to show that types are preserved during evaluation, it is necessary to have a semantic object which associates dynamically allocated channel identifiers with appropriate types.

Definition 3.8 (Channel environment). A channel environment is a finite map from channel identifiers to type and mobility annotation pairs. $CE ::= [] \mid CE[k \mapsto (\tau,\mu)]$. The type and the mobility annotation represent respectively the type of values communicated by the channel and the mobility annotation of the channel.

Definition 3.9 (Extension).

- Let Γ and Γ' be two type environments. Γ' extends Γ, written as $\Gamma \sqsubseteq \Gamma'$ if $Dom(\Gamma) \subseteq Dom(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for all $x \in Dom(\Gamma)$.
- Let CE and CE' be two channel environments. CE' extends CE, written as $CE \sqsubseteq CE'$ if $Dom(CE) \subseteq Dom(CE')$ and $CE(k) = CE'(k)$ for all $k \in Dom(CE)$.

Definition 3.10 (Compatible environments). We call two typing environments compatible if they agree on the types of ubiquitous values. Let Γ and Γ' be two type environments. Γ' is compatible with Γ, written as $\Gamma \sim \Gamma'$ if $\Gamma(x) = \Gamma'(x)$ for all $x \in U$.

3.5.1 Types for intermediate expressions

The proof of type preservation property requires us to be able to type the intermediate expressions which we have introduced in Section 3.3.2. It is clear that every expression is also an intermediate expression. Therefore, the typing rules of Figures 3.5 and 3.6

can be regarded as typing rules for intermediate expressions which conform to the abstract syntax of Mobile-λ. An addditional set of typing rules is presented below for the remaining forms of intermediate expressions. The typing judgements for intermediate expressions include CE as well as Γ. In general, we do not write CE explicitly if there is no scope for confusion.

$CE,\Gamma \vdash k : \tau, \emptyset$ where $k = (l,s,i)$	if $CE(k) = (\tau,\mu)$ and $l \subseteq \mu$
$CE,\Gamma \vdash \langle l,E,x,e\rangle : (\tau \xrightarrow{\kappa} \tau',\mu), \emptyset$	if there exists a Γ' such that $\Gamma \sim \Gamma'$ and $\Gamma' \vdash E$
	and $\Gamma' \vdash \mathsf{fn}^l\, x \Rightarrow e : (\tau \xrightarrow{\kappa} \tau',\mu)$
$CE,\Gamma \vdash \mathsf{bind}\, E \;\mathsf{in}\; ie : \tau, \kappa$	if there exists a Γ' such that $\Gamma \sqsubseteq \Gamma'$
	and $\Gamma' \vdash E$ and $\Gamma' \vdash ie : \tau, \kappa$
$CE,\Gamma \vdash v : \forall \vec{\alpha}\vec{\beta}\vec{\gamma}.\tau, \emptyset$	if $\Gamma \vdash v : \theta\tau$ for any θ defined on $\vec{\alpha}\vec{\beta}\vec{\gamma}$
$CE,\Gamma \vdash E$	if $Dom(\Gamma) = Dom(E)$
	and $\Gamma \vdash E(x) : \Gamma(x)$ for all $x \in Dom(E)$

3.5.2 Type soundness

The following lemma which shows that typing is stable under substitution is used in the proof of the type preservation theorem.

Lemma 3.1 (Substitution). If $\Gamma \vdash ie : \tau, \kappa$ then $\theta\Gamma \vdash ie : \theta\tau, \theta\kappa$ for any substitution θ.

Proof. The proof is given by induction on the depth of derivation of $\Gamma \vdash ie : \tau, \kappa$. Two selected cases of the proof are presented in the Appendix. $\square$

Lemma 3.2 (Subsumption elimination). We can assume for the derivation of a typing judgement $CE,\Gamma \vdash ie : \tau, \kappa$ that the non-structural rule (subs) is used after every structural rule.

Proof. Any derivation tree for $CE,\Gamma \vdash ie : \tau, \kappa$ can be transformed into a derivation tree where we use the rule (subs) after every structural rule. By transitivity of the subset relation $\subseteq$ on sets we can eliminate multiple applications of the (subs) rule and consider it as a single application of the rule. $\square$

Theorem 3.1 (Type preservation). Let CE be a global channel environment, E_s and Γ_s denote the evaluation and type environments at site s. Assume

- $\Gamma_{s_1} \sim \Gamma_{s_2}$ for all site pairs s_1, s_2 and $\Gamma_s \vdash E_s$ for all s in the system
- $CI, P[(s_1, E_{s_1}) : ie_1][(s_2, E_{s_2}) : ie_2] \xrightarrow{a} CI', P'[(s_1, E_{s_1}) : ie_1'][(s_2, E_{s_2}) : ie_2']$ where all of the channel identifiers which occur in ie_i are in CI for $i \in \{1,2\}$ and $CI \subseteq Dom(CE)$.
- $\Gamma_1 \vdash ie_1 : \tau_1, \kappa_1$ and $\Gamma_2 \vdash ie_2 : \tau_2, \kappa_2$ where all of the free variables which occur in ie_i are in $Dom(\Gamma_i)$ for $i \in \{1,2\}$.

Then the following hold:

- if $a = \varepsilon$ then $\Gamma_1 \vdash ie_1' : \tau_1, \kappa_1$ and $\Gamma_2 \vdash ie_2' : \tau_2, \kappa_2$
- if $a = s_i[\mathsf{new}\, k]$ then $\Gamma_i \vdash ie_i' : \tau_i, \kappa_i$ and there exists a τ,μ and CE' such that $CE \sqsubseteq CE'$ and $CE'(k) = (\tau,\mu)$ and $CI' \subseteq Dom(CE')$ and $\{\mathtt{new}\mu\ \mathsf{for}\ \tau\} \subseteq \kappa_i$ for $i \in \{1,2\}$.
- if $a = (s_1[k!v], s_2[k?v])$ then $\Gamma_1 \vdash ie_1' : \tau_1, \kappa_1$ and $\Gamma_2 \vdash ie_2' : \tau_2, \kappa_2$ and there exists a τ,μ such that $\Gamma_1 \vdash v : \tau, \emptyset$ and $CE(k) = (\tau,\mu)$ and $\{\mathsf{send}\,\tau\,\mathsf{on}\,\mu\} \subseteq \kappa_1$ and $\{\mathsf{recv}\,\tau\,\mathsf{on}\,\mu\} \subseteq \kappa_2$.

Proof. The proof is given by induction on the depth of derivation of

$$CI, P[(s_1, E_{s_1}) : ie_1][(s_2, E_{s_2}) : ie_2] \xrightarrow{a} CI, P[(s_1, E_{s_1}) : ie_1'][(s_2, E_{s_2}) : ie_2'].$$

For the cases which involve no communication, it is obvious that the transition involves only one of the expressions. Therefore, it is sufficient to prove the theorem considering one of the expressions only. Selected cases of the proof are shown below.

case $ie = x$. The dynamic semantics requires the evaluation to follow the rule (id), that is $CI, P[(s,E) : x] \xrightarrow{\varepsilon} CI, [(s,E) : E(x)]$. Suppose $\Gamma \vdash x : \tau, \emptyset$. The typing rule (id) forces that $\Gamma(x) = \sigma$ where $\sigma \succ \tau$. We know by the assumptions that $\Gamma \vdash E$. This also implies that $\Gamma \vdash E(x) : \Gamma(x)$. Let $\Gamma(x) = \sigma = \forall\vec{\delta}.\tau_x$. By the definition of value typing it must be the case that $\Gamma \vdash E(x) : \theta\tau_x$ for any θ defined on $\vec{\delta}$. Then $\Gamma \vdash E(x) : \tau, \emptyset$ since

$\sigma \succ \tau$.

case $ie = (\mathsf{fn}^l\, x \Rightarrow e)$. The dynamic semantics requires the evaluation to follow the rule (fn), that is $CI, P[(s,E) : \mathsf{fn}^l\, x \Rightarrow e] \xrightarrow{\varepsilon} CI, P[(s,E) : \langle l, E', x, e\rangle]$ where $E' = E \downharpoonright FI(\mathsf{fn}^l\, x \Rightarrow e)$. Suppose $\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e : (\tau \xrightarrow{\kappa} \tau', \{l\} \cup \mu), \emptyset$. The typing rule (fn) tells us that for this judgement to hold it must be the case that $\Gamma[x \mapsto \tau] \vdash e : \tau', \kappa$ where $\mu = M(\Gamma, FI(\mathsf{fn}^l\, x \Rightarrow e))$.

We know by the assumption that $\Gamma \vdash E$. This implies $\Gamma \vdash E'$ since E' is a restriction of E. We can conclude that there exists a Γ' such that $\Gamma' \vdash \langle l, E', x, e\rangle : (\tau \xrightarrow{\kappa} \tau', l \cup \mu), \emptyset$ and by taking Γ as a witness for Γ' and using the typing rule for function closures.

case $ie = (\mathsf{let}\; x = v\, \mathsf{in}\; e)$. The dynamic semantics requires the evaluation to follow the rule (let-3), that is $CI, P[(s,E) : \mathsf{let}\; x = v\; \mathsf{in}\, e] \xrightarrow{\varepsilon} CI, P[(s,E) : \mathsf{bind}\, E'[x \mapsto v]\, \mathsf{in}\, e]$ where $E' = E \downharpoonright FI(e)$. Suppose $\Gamma \vdash \mathsf{let}\; x = v\; \mathsf{in}\, e : \tau', \kappa \cup \kappa'$. The typing rule (let) requires that $\Gamma \vdash v : \tau, \kappa$ and $\Gamma[x \mapsto Gen(\Gamma, \kappa, \tau)] \vdash e : \tau', \kappa'$.

By Lemma 3.2 we know that there exist κ_1 and κ_2 such that $(\kappa_1 \cup \kappa_2) \subseteq (\kappa \cup \kappa')$ where $\Gamma \vdash v : \tau, \kappa_1$ and $\Gamma[x \mapsto Gen(\Gamma, \kappa, \tau)] \vdash e : \tau', \kappa_2$. By Lemma 3.1 we have $\theta\Gamma \vdash v : \theta\tau, \theta\kappa_1$ for any substitution θ. For θ defined on those variables which are not free in Γ or κ_1 it is the case that $\Gamma \vdash v : \theta\tau, \kappa_1$. Let these variables be $\vec{\delta}$. The typing rules for values allows us to conclude that $\Gamma \vdash v : \forall\vec{\delta}.\tau, \kappa_1$. This is equivalent to saying that the type of v is $Gen(\Gamma, \kappa_1, \tau)$. We take a Γ' such that $\Gamma' = \Gamma[x \mapsto Gen(\Gamma, \kappa_1, \tau)]$. We can conclude that $\Gamma' \vdash E'[x \mapsto v]$. Given $\Gamma[x \mapsto Gen(\Gamma, \kappa_1, \tau)] \vdash e : \tau', \kappa_2$, the typing rule for bind expressions allow us to conclude that $\Gamma, CE \vdash \mathsf{bind}\, E'[x \mapsto v]\, \mathsf{in}\, e : \tau', \kappa_2$. By Lemma 3.2 $\mathsf{bind}\, E'[x \mapsto v]\, \mathsf{in}\, e : \tau', \kappa \cup \kappa'$.

case $ie = \mathsf{chan}^l()$. The dynamic semantics requires the evaluation to follow the rule (chan), that is $CI, P[(s,E) : \mathsf{chan}^l()] \xrightarrow{s[\mathsf{new}\, k]} CI \cup k, P[(s,E) : k]$. Suppose $\Gamma \vdash \mathsf{chan}^l() : (\mathsf{chan}[\tau], l), \{\mathtt{new}\, l \text{ for } \tau\}$. By the typing rule (chan) we know that $k \notin CI$. Taking a channel environment such that $CE' = CE[k \mapsto (\tau, l)]$ for some τ satisfies all the necessary conditions.

case $ie_1 = (k!v)$ and $ie_2 = (k?)$. The dynamic semantics requires the evaluation to follow the rule (com), that is $CI, P[(s_1, E_1) : k\,!\,v][(s_2, E_2) : k?] \xrightarrow{a} CI, P[(s_1, E_1) : ()][(s_2, E_2) : v]$ where $a = (s_1[k!v], s_2[k?v])$. Suppose $\Gamma \vdash k!v : \mathsf{unit}, \kappa \cup \kappa' \cup \{\mathsf{send}\,\tau\,\mathsf{on}\,\mu\}$ and $\Gamma \vdash k? : \tau, \kappa \cup \{\mathsf{recv}\,\tau\,\mathsf{on}\,\mu\}$. By the typing rules (send) and (receive) we know that $\Gamma_1 \vdash k : (\mathsf{chan}[\tau], \mu), \kappa$ and $\Gamma_1 \vdash v : \tau, \kappa'$ and $\Gamma_2 \vdash k : (\mathsf{chan}[\tau], \mu), \kappa$. This case refers to Lemma 3.2 as the case above. The rest of the proof is immediate by a simple inspection of the typing rules for receive and send expressions. □

3.5.2.1 Run-time errors

Definition 3.11 (Run-time error). The evaluation of an intermediate expression *ie* causes a run-time error, written as $CI, P[(s, E) : ie] \longrightarrow \mathsf{ERROR}$ if all of the following propositions are simultaneously true:

- *ie* is not a value;
- *ie* is not blocked on communication; and
- there is no evaluation rule such that $CI, P[(s, E) : ie] \longrightarrow CI', P'[(s, E) : ie']$.

Note that the above characterization of a run-time error does not capture the errors which are caused by the absence of a communication partner. For example, an expression can attempt at run-time to send a value on a channel which is listened to by no other process. We do not treat this as a run-time error. However, any attempt to send a value which does not match with the type of the channel would cause a run-time error.

Theorem 3.2 (Type soundness). Let Γ and E be the type environment and evaluation environment at site s. Let CE be a global channel environment and CI be a global channel identifier set. Assume $CE, \Gamma \vdash ie : \tau, \kappa$ and $CE, \Gamma \vdash E$ and $CI \subseteq Dom(CE)$. Then it is not the case that $CI, P[(s, E) : ie] \longrightarrow \mathsf{ERROR}$.

Proof. The proof is given by induction on the depth of the typing derivation of *ie*. □

3.5.3 Principal typing

Given a typing context and an expression, the typing rules presented above do not necessarily assign a unique type to an expression. It could be possible to derive zero, one or even infinitely many types for an expression. The theorem below shows that if an expression can be typed at all in our system it also has a *principal* type, that is a type which is the most general type with respect to substitution on variables. Similarly, there is an effect which is minimal with respect to the subset relation on sets.

Theorem 3.3 (Principal Typing). If a type can be derived for an expression e in the type system then there exists an environment $\theta^p\Gamma$, a type τ^p and κ^p such that $\theta^p\Gamma \vdash e : \tau^p, \kappa^p$ and whenever $\theta\Gamma \vdash e : \tau, \kappa$ then for some substitution ψ it is the case that $\psi(\theta^p\Gamma) = \theta\Gamma$ and $\psi\tau^p = \tau$ and $\kappa \subseteq \psi\kappa^p$. The type τ^p is principal for e in Γ.

Proof. Selected proof cases can be found in the Appendix. □

3.6 Static estimation

As we provided explanations on our particular choice of annotations in Section 3.4, at this point it is relatively clear how we intend to use the static information captured in types to estimate mobile functions and channels. Our approach is as follows. Supposing that an expression is associated with an effect κ we consider each $\{\mathsf{send}\,\tau\,\mathsf{on}\,\mu\} \subseteq \kappa$ in isolation, analyse τ to collect the mobility annotations (labels) of interest and finally merge the results.

3.6.1 Extracting labels

Definition 3.12 ($\models$). We define the relation $\models$ as the smallest relation which satisfies the rules of Figure 3.7. It relates types and effects to sets of labels. The relation for types and the relation for effects are defined mutually recursively. A set of labels L associated with an effect κ estimates the labels of values which become mobile when an expression with effect κ is executed.

Rules 1, 2 and 3 for types apply when the raw type is a base type or a variable. Our type system does not keep track of the values of base types and a variable type does not contain any specific information that we need to note. Therefore, we take L to be empty. In the case of channel types, however, we collect the mobility annotation as shown in Rule 4.

	Types		**Effects**
(t1)	$\models (\mathsf{unit},\mu) : \emptyset$	**(e1)**	$\models \emptyset : \emptyset$
(t2)	$\models (\mathsf{int},\mu) : \emptyset$	**(e2)**	$\models \{\mathsf{new}(\bar{\tau},\mu)\} : \emptyset$
(t3)	$\models (\alpha,\mu) : \emptyset$	**(e3)**	$\models \beta : \emptyset$
(t4)	$\models (\mathsf{chan}[\tau],\mu) : \mu$	**(e4)**	$\models \{\mathsf{recv}\,\tau\,\mathsf{on}\,\mu'\} : \emptyset$
(t5)	$\dfrac{\models \tau' : L \quad \models \kappa : L'}{\models (\tau_1 \xrightarrow{\kappa} \tau',\mu) : \mu \cup L \cup L'}$	**(e5)**	$\dfrac{\models \tau : L}{\models \{\mathsf{send}\,\tau\,\mathsf{on}\,\mu'\} : L}$
		(e6)	$\dfrac{\models \kappa : L \quad \models \kappa' : L'}{\models \kappa \cup \kappa' : L \cup L'}$

Figure 3.7: Mobility Analysis

Rule 5 applies when the type is a function type. As well as extracting the mobility annotation of the function, the result type and the effect are also examined to collect the annotation of those values which may escape to another site and be sent from that site subsequently. Note that the need for this has been motivated by Example 3.3 in Section 3.2.1.

The rules for effects show that it is the send effects which we are concerned with. The types of the sent values are examined to estimate potentially mobile values.

3.6.2 Soundness

We have shown that our type system associates values with types which are consistent in the sense defined in Section 3.5.2. The soundness of the analysis presented above relies on this fact. The definition of the notion of soundness for the mobility analysis states that the analysis of a correctly typed expression would conservatively estimate the labels of interest. Conservative estimation is expressed by the subset relation.

Lemma 3.3 (Soundness of label estimation).

$$\text{If } \Gamma \vdash v : \tau, \emptyset \text{ and } \models \tau : L \text{ then } \mathit{Labels}(v) \subseteq L.$$

Proof. The proof is given by considering all possible forms of v. We give the proof of two selected cases only.

case $v = k = (s,l,i)$. By the hypothesis, we have $CE,\Gamma \vdash k : (\mathsf{chan}[\tau],\mu),\emptyset$ where $l \subseteq \mu$. By Rule t4 of Figure 3.7 we also know that $\models (\mathsf{chan}[\tau],\mu) : \mu$. By Definition 3.4 $Labels(k) = \{l\}$. Since $L = \mu$ and $l \subseteq \mu$ it follows that $Labels(v) \subseteq L$.

case $v = \langle l,E,x,e\rangle$. By the hypothesis we have $CE,\Gamma \vdash \langle l,E,x,e\rangle : (\tau \xrightarrow{\kappa} \tau',\mu)$. This requires there to exist a Γ' such that $\Gamma \sim \Gamma'$ and $\Gamma' \vdash E$ and $\Gamma' \vdash \mathsf{fn}^l\, x \Rightarrow e : (\tau \xrightarrow{\kappa} \tau',\mu),\emptyset$. The typing rule for functions allows us to conclude also that $\Gamma'[x \mapsto \tau] \vdash e : \tau',\kappa$ where $\mu = \{l\} \cup M(\Gamma', FI(\mathsf{fn}^l\, x \Rightarrow e))$. By Definition 3.4 we have $Labels(\langle l,E,x,e\rangle) = \{l\} \cup S'$ where $S' = \bigcup\{Labels(E(x)) \mid x \in Dom(E)\}$. By Rule t5 of Figure 3.7 we also have $\models (\tau \xrightarrow{\kappa} \tau',\mu) : \mu \cup L \cup L'$ where $\models \tau' : L$ and $\vdash \kappa : L'$. Since $\Gamma' \vdash E$ we have $\Gamma' \vdash E(x) : \Gamma'(x)$ for every $x \in Dom(E)$. By referring to Definition 3.7 we can conclude that $S' \subseteq M(\Gamma', FI(\mathsf{fn}^l\, x \Rightarrow e))$. Hence $(\{l\} \cup S') \subseteq \mu$. It follows that $Labels(v) \subseteq (\mu \cup L \cup L')$ □

Theorem 3.4 (Soundness of analysis). Consider a single step in the evaluation of an expression *ie* such that

- $CI,P[(s,E) : ie] \xrightarrow{a} CI,P[(s,E) : ie']$ where $a = (s[k!v], s'[k?v])$
- $\Gamma \vdash E$ and $\Gamma \vdash ie : \tau,\kappa$ where $\models \kappa : L$

Let $Mobile = Labels(v)$. Then $Mobile \subseteq L$.

Proof. The proof is given by straightforward induction on the derivation of evaluation. □

3.7 Concluding Remarks

In this chapter we have shown that the methodology of annotated type and effect systems can be exploited to predict the mobility of values in a higher-order functional language extended with primitives for communication. We have included in the language Mobile-λ only those features which are crucial for our investigation. For example, Mobile-λ can be extended with features for dynamic process creation at local or remote sites. This does not, however, provide new insights into the problem.

In Mobile-λ any communication is assumed to take place between two remote parties. Therefore any communication gives rise to mobility between two remote sites. Consequently, the definitions of the dynamic and static semantics treat any communication as observable. If we allowed dynamic creation of processes at local and remote sites it would be appropriate to define an observation criterion such as the ones in [Tho94, TJ94] to distinguish between local and remote communications. The type system then would have to trace the annotations of remotely communicated values only. Such an extension can easily be incorporated into the existing type system.

Devising an algorithm to infer the principal types and minimal effects is out of the scope of this chapter. However, the existing works in the field suggest that such an algorithm can be devised and implemented.

Chapter 4

Distributed Call-Tracking

One of the positive qualities associated with mobile computation is that it provides a flexible setting for sharing computational resources. A piece of code can move towards a site which hosts the resources it aims to exploit. Input/output devices, file systems, network, memory and processing power (CPU cycles) are typical examples of shared resources which may be exploited by mobile code.

In a system where shared resources such as processing power and memory are limited, it is particularly important that some programs do not exhaust the resources at the expense of other programs. For example, if access to CPU cycles is not controlled to enforce an appropriate fairness criterion, a program can hinder the execution of another program which uses the same processor, by occupying the CPU most of the time. Similarly, if a program is allowed to allocate unlimited memory, it may cause the system to deny service to other programs.

Computation in active networks with the Programming Language for Active Networks (PLAN), introduced in Chapter 2, is a specific instance of mobile computation in a system where the allocation of resources to programs must be strictly controlled. Denial of service attacks are considered as one of the major threats to the safety and security of active networks. The design of PLAN has been highly influenced by the need to prevent denial of service.

The ideas underlying the design of PLAN could form the basis of a general purpose mobile code language which is suitable for computation in resource-sensitive systems if the restrictions of PLAN could be made less severe. In this chapter, we introduce a language which mitigates some of the restrictions in PLAN's computational model and extends it with support for mobility through remote evaluation of functions in the spirit of the model proposed by Stamos and Gifford [SG90].

In an applicative language where function applications constitute the basis of computation, attempts to estimate the resource consumption of programs would benefit from the static analysis of control flow within a program. For example, if we take processing power as the resource of interest, the number of functions called is an indicator for the processing time demanded by the evaluation of a program. In case the language facilitates remote evaluation of functions on different sites, the estimation of processing time demanded from a particular site requires the estimation of functions called at that site.

In this chapter, we investigate the use of annotated type systems in estimating which functions are called at which sites. We call this *distributed call-tracking analysis*. Our approach can be regarded as extending the work on type-based *call-tracking analysis* for sequential functional languages [TJ92, Hei94a]. In addition to the potential uses we have mentioned above, the information obtained by distributed call-tracking analysis can serve as the starting point for traditional compiler optimizations as well as allowing compilers to produce code optimized for particular sites which a function may visit at run-time.

4.1 Security through language restrictions

In Chapter 2 we have presented the essential elements of the design of PLAN. In this section we recapitulate those features of PLAN which are motivated by the need to prevent denial-of-service attacks and to prevent programs from interfering with one another's execution.

4.1.1 Termination and resource bounds

Non-terminating programs consume the resources of a system unboundedly and therefore make them vulnerable to denial-of-service attacks. The designers of PLAN have adopted a strict approach in dealing with this problem. They have ensured that pure

PLAN programs terminate by eliminating recursion and non-fixed length iteration, and imposing resource bounds on the number of packets generated. The following example from [HK99] motivates the need for explicit resource bounds in PLAN. Without a resource bound counter which is decremented each time a new packet is generated, the function `ping_pong` below would move between the two nodes of the network forever.

```
fun ping_pong  (pingHost:host, pongHost:host):unit =
    OnRemote   (|ping_pong (pongHost, pingHost)|,
               pongHost, getRB (), defaultroute)
```

It is argued in [HK99] that even these restrictions are not adequate and it would be more appropriate to ensure that PLAN programs contained in a packet require bandwidth, CPU and memory linear in the size of the packet. A detailed discussion about the motivating factors and the alternative approaches for implementing this can be found in [HK99, MHN01]. The execution of a function such as `exponential` below is considered to be undesirable because the number of function calls is exponential in the number of function definitions.

```
fun f1()= ()
fun f2()= (f1();f1())
fun f3()= (f2();f2())
fun f4()= (f3();f3())
fun exponential()=(f4();f4())
```

4.1.2 Isolation and strong typing

Pure PLAN programs are executed in isolation from one another. There is no notion of shared mutable state or communication channels as in Facile or Concurrent ML. No language mechanism for direct communication exists. This ensures that programs have their own logical space for data and there can be no interference between them.

PLAN is a strongly typed language. The combination of strong typing and automatic memory management proves useful in ruling out indirect means of communication, for example, due to buffer overflows [LR98, HK99].

4.1.3 Exploiting static analysis

It is stated by the authors who work on the formal specification of PLAN [KHMG99] that PLAN programs should be type-checked at each node where they are executed. The capability for static type-checking is seen as a requirement while dynamic type-checking is allowed as an option. There is no doubt that strong static typing is a crucial security property for a PLAN-like language. We observe that if the type system could be exploited further to infer information about the function call behaviour of programs it would be possible to enhance the advantages brought about by strong typing. Instead of ruling out recursion from the language and imposing resource bounds, one could adopt a more liberal approach. Each site could enforce its own policy to protect itself from the potential threats posed by a program. This would be particularly appealing for computation in contexts where resources are not as scarce as in active networks.

For example, if the type system could expose that the function `ping_pong` above calls itself at two distinct sites, this could be taken as an indication for a possibly non-terminating computation which affects two sites. In case the system is highly vulnerable to computations of this nature, the execution of the program could be disallowed. Likewise, if functions such as `exponential` are undesirable, exposing the number of function calls initiated by `exponential` would be sufficient to alarm the type-checking site about the potential risks.

In the following sections we present two type systems for distributed-call tracking analysis. The first one is a monomorphic type system which uses sets to estimate which functions are called at which sites. The second type system incorporates a limited form of polymorphism and captures finer-grained information about the control flow in programs.

4.2 rEval-λ

In this section we introduce a language for remote evaluation – rEval-λ – which extends the Core language of Chapter 2 with site names, recursive expressions and primitives for initiating the remote evaluation of expressions.

4.2.1 Abstract syntax

As in Mobile-λ from Chapter 3, all functions are labelled. A predefined collection of site names is available for rEval-λ expressions to specify the sites in a system. Additionally, rEval-λ provides two constructs for sending an expression to be evaluated at a remote site. These are distinguished by the way in which the result of the evaluation is handled. We provide a more detailed explanation about this in the following section.

Site names	s	::=	$s_1 \mid s_2 \mid \dots$	
Function labels	l	::=	$l_1 \mid l_2 \dots$	
Expressions	e	::=	$\dots$	expressions of the Core
			$\mid s$	site names
			$\mid \mathsf{fn}^l\, x \Rightarrow e$	function abstraction
			$\mid \mathsf{rec}^l\, f(x) \Rightarrow e$	recursive functions
			$\mid \mathsf{reval}\,(e_1, e_2)\,\mathsf{at}\, e_3$	remote evaluation
			$\mid \mathsf{spawn}\,(e_1, e_2)\,\mathsf{at}\, e_3$	remote spawn

Figure 4.1: Abstract Syntax for rEval-λ

4.2.2 Dynamic semantics

We model a system as a collection of sites where a site may host the execution of multiple threads of control. Sites are uniquely identified by site names. Values of the language consist of basic constants, site names and functions.

$$\textbf{Values} \quad v \quad ::= \quad c \mid s \mid \mathsf{fn}^l\, x \Rightarrow e$$

We assume that each site in the system can run rEval-λ programs and provides a standard set of services. These standard services are referred to as *ubiquitous values*. Each site provides its local implementation for a ubiquitous value and bindings to these values are created dynamically.

The semantic definition of remote evaluation requires expressions which denote blocked computations. We introduce the expressions of the form $\mathsf{block}_{(s,p)}\,\mathsf{on}\,e$ for this purpose. The expressions of the dynamic semantics include blocked expressions as well as expressions of the forms which are presented in Figure 4.1.

4.2.2.1 Evaluation rules

A system state S is represented as a set of tuples written as $[(s,p):e]$. A tuple $[(s,p):e]$ represents a process p at site s which is currently executing the expression e. The notation $S[(s,p):e]$ is used to denote the set $S \cup [(s,p):e]$. To simplify presentation we assume that expressions do not contain free variables.

A single step transition from state S to state S' is represented by a judgement of the form $S \xrightarrow{a} S'$. The annotation a on the arrow records the flow of control during the transition. The type systems we present in the following sections estimate the functions called by a program. By making function calls observable in the semantics in the form of annotations, we prepare the ground for proving the soundness of our type systems in the later sections.

Actions	a	$::=$	ε	no function call
			$\mid\ l@s$	call function l at site s

Rules of Figure 4.2 These rules are similar to the evaluation rules we have presented for the Core and the sequential subset of Mobile-λ. The only rule which appears for the first time is the rule (rec) for recursive functions. It shows the one step unfolding of a recursive function. The label of the recursive function is carried over to the function abstraction which is obtained by the unfolding.

Rules of Figure 4.3 The evaluation rules which involve distributed computation are different from the evaluation rules we have considered so far. There are five rules concerning a remote evaluation expression. The first three of these rules, (reval-1)

(rec) $S[(s,p) : \mathsf{rec}^l\, f\,(x) \Rightarrow e] \xrightarrow{\varepsilon} S[(s,p) : \mathsf{fn}^l\, x \Rightarrow e\{(\mathsf{rec}^l\, f\,(x) \Rightarrow e)/f\}]$

(app-1) $$\frac{S[(s,p) : e_1] \xrightarrow{a} S'[(s,p) : e_1']}{S[(s,p) : e_1\, e_2] \xrightarrow{a} S'[(s,p) : e_1'\, e_2]}$$

(app-2) $$\frac{S[(s,p) : e_2] \xrightarrow{a} S'[(s,p) : e_2']}{S[(s,p) : v\, e_2] \xrightarrow{a} S'[(s,p) : v\, e_2']}$$

(app-3) $S[(s,p) : (\mathsf{fn}^l\, x \Rightarrow e)\, v] \xrightarrow{l@s} S[(s,p) : e\{v/x\}]$

(op-1) $$\frac{S[(s,p) : e_1] \xrightarrow{a} S'[(s,p) : e_1']}{S[(s,p) : e_1\ op\ e_2] \xrightarrow{a} S'[(s,p) : e_1'\ op\ e_2]}$$

(op-2) $$\frac{S[(s,p) : e_2] \xrightarrow{a} S'[(s,p) : e_2']}{S[(s,p) : v\ op\ e_2] \xrightarrow{a} S'[(s,p) : v\ op\ e_2']}$$

(op-3) $S[(s,p) : v_1\ op\ v_2] \xrightarrow{\varepsilon} S[(s,p) : v]$ where $v = v_1\ \mathbf{op}\ v_2$

(if-1) $$\frac{S[(s,p) : e_1] \xrightarrow{a} S'[(s,p) : e_1']}{S[(s,p) : \mathsf{if}\ e_1 \mathsf{then}\ e_2\ \mathsf{else}\ e_3] \xrightarrow{a} S'[(s,p) : \mathsf{if}\ e_1' \mathsf{then}\ e_2\ \mathsf{else}\ e_3]}$$

(if-2) $S[(s,p) : \mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3] \xrightarrow{\varepsilon} S[(s,p) : e_2]$

(if-3) $S[(s,p) : \mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3] \xrightarrow{\varepsilon} S[(s,p) : e_3]$

(let-1) $$\frac{S[(s,p) : e_1] \xrightarrow{a} S'[(s,p) : e_1']}{S[(s,p) : \mathsf{let}\ x = e_1 \mathsf{in}\ e_2] \xrightarrow{a} S'[(s,p) : \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2]}$$

(let-2) $S[(s,p) : \mathsf{let}\ x = v\ \mathsf{in}\ e_2] \xrightarrow{\varepsilon} S[(s,p) : e_2\{v/x\}]$

Figure 4.2: Sequential Evaluation Rules

to (reval-3), serve the purpose of specifying the evaluation order; subexpressions are evaluated in left-to-right order as is the case for all forms of expressions. Once all the subexpressions are fully evaluated, the evaluation can proceed only if the first expression is a function and the third one is a site name.

Before explaining the rule (reval-4), we need to clarify what an intermediate expression of the form $\mathsf{block}_{(s,p)}\,\mathsf{on}\,e$ represents. Intermediate expressions of this form are introduced to capture suspended computations. For example, the state $S[(s,p) : \mathsf{block}_{(s',p')}\,\mathsf{on}\,e]$ indicates that the evaluation of process p at site s is waiting for the value of e which will be returned by the process p' at site s'. The rule (reval-4) shows that the function body is sent for evaluation at the specified site with the value of the second expression used as the actual argument. This causes the evaluation at the sending site to enter a blocked state. The rule (reval-5) shows that the evaluation can only resume when the computation at the remote evaluation site returns a value. This value is taken to be the value of the blocked expression. Note that our dynamic semantics abstracts from how the communication between two remote sites occurs. The only observable action is the function call at the remote site. We assume that a new process is generated each time a remote evaluation is initiated. It is implicit in our rules that dynamic generation of processes at the local site is a special case of the remote evaluation where $s = s'$.

The evaluation of remote spawn expressions is similar except that the expression which initiates the evaluation at a remote site does not wait for the result to be sent back to it; it returns a unit value immediately. This implies that spawn expressions would typically be used for their side effects.

4.2.3 Examples

In this section we present three examples of small rEval-λ programs. The first one illustrates the use of the remote evaluation facility for a simple but useful purpose. The second example illustrates that an improper use of the remote evaluation facility can lead to undesirable exploitation of processing power. The third example illustrates that the application of some functions can cause a quick growth in the number of subsequent function calls.

(reval-1)
$$\frac{S[(s,p):e_1]\xrightarrow{a}S'[(s,p):e_1']}{S[(s,p):\mathsf{reval}\,(e_1,e_2)\,\mathsf{at}\,e_3]\xrightarrow{a}S'[(s,p):\mathsf{reval}\,(e_1',e_2)\,\mathsf{at}\,e_3]}$$

(reval-2)
$$\frac{S[(s,p):e_2]\xrightarrow{a}S'[(s,p):e_2']}{S[(s,p):\mathsf{reval}\,(v,e_2)\,\mathsf{at}\,e_3]\xrightarrow{a}S'[(s,p):\mathsf{reval}\,(v,e_2')\,\mathsf{at}\,e_3]}$$

(reval-3)
$$\frac{S[(s,p):e_3]\xrightarrow{a}S'[(s,p):e_3']}{S[(s,p):\mathsf{reval}\,(v_1,v_2)\,\mathsf{at}\,e_3]\xrightarrow{a}S'[(s,p):\mathsf{reval}\,(v_1,v_2)\,\mathsf{at}\,e_3']}$$

(reval-4)
$$S[(s,p):\mathsf{reval}\,((\mathsf{fn}^l\,x\Rightarrow e),v)\,\mathsf{at}\,s']\xrightarrow{l@s'}$$
$$S[(s,p):\mathsf{block}_{(s',p')}\,\mathsf{on}\,e\{v/x\}][(s',p'):e\{v/x\}] \text{ where } p' \text{ new at } s'$$

(reval-5)
$$S[(s,p):\mathsf{block}_{(s',p')}\,\mathsf{on}\,e][(s',p'):v]\xrightarrow{\varepsilon}S[(s,p):v]$$

(spawn-1)
$$\frac{S[(s,p):e_1]\xrightarrow{a}S'[(s,p):e_1']}{S[(s,p):\mathsf{spawn}\,(e_1,e_2)\,\mathsf{at}\,e_3]\xrightarrow{a}S'[(s,p):\mathsf{spawn}\,(e_1',e_2)\,\mathsf{at}\,e_3]}$$

(spawn-2)
$$\frac{S[(s,p):e_2]\xrightarrow{a}S'[(s,p):e_2']}{S[(s,p):\mathsf{spawn}\,(v,e_2)\,\mathsf{at}\,e_3]\xrightarrow{a}S'[(s,p):\mathsf{spawn}\,(v,e_2')\,\mathsf{at}\,e_3]}$$

(spawn-3)
$$\frac{S[(s,p):e_3]\xrightarrow{a}S'[(s,p):e_3']}{S[(s,p):\mathsf{spawn}\,(v_1,v_2)\,\mathsf{at}\,e_3]\xrightarrow{a}S'[(s,p):\mathsf{spawn}\,(v_1,v_2)\,\mathsf{at}\,e_3']}$$

(spawn-4)
$$S[(s,p):\mathsf{spawn}((\mathsf{fn}^l\,x\Rightarrow e),v)\,\mathsf{at}\,s')]\xrightarrow{l@s'}$$
$$S'[(s,p):()][(s',p'):e\{v/x\}] \text{ where } p' \text{ new at } s'$$

Figure 4.3: Distributed Evaluation Rules

Example 4.1. We assume that $\mathsf{getTime}$ is a ubiquitous function which takes a unit argument. An expression which returns the local time at a remote site $\mathsf{s1}$ can be coded in rEval-λ as follows.

$$\begin{array}{rll} e = & \mathsf{let} & \mathsf{remoteTime} \ = \ \mathsf{fn}^{l}\,\mathsf{x} \Rightarrow \mathsf{reval}(\mathsf{getTime},()) \ \mathsf{at}\ \mathsf{x} \\ & \mathsf{in} & \mathsf{remoteTime}\ \mathsf{s1} \end{array}$$

Example 4.2. If $\mathsf{thisHost}$ is a ubiquitous function which returns the name of the site where the program is currently running, the evaluation of the following expression at site $\mathsf{s0}$ would trigger repeated invocations of f on sites $\mathsf{s0}$ and $\mathsf{s1}$. Note that the effect of this expression can be likened to that of the PLAN function $\mathsf{ping\_pong}$ presented in Section 4.1.

$$\begin{array}{rll} e = & \mathsf{let} & \mathsf{f} = \mathsf{rec}^{l}\mathsf{f}(\mathsf{x}) \Rightarrow \mathsf{spawn}(\mathsf{f},\mathsf{thisHost}())\ \mathsf{at}\ \mathsf{x} \\ & \mathsf{in} & \mathsf{f}\ \mathsf{s1} \end{array}$$

Example 4.3. The function twice is a higher-order function which takes two arguments, the first of which is a function. The full evaluation of twice results in its first argument being applied twice, once to its second argument and once to the result of its application to the second argument. The code presented below shows that the application of twice to the function $\mathsf{f3}$ and $()$ gives rise to several nested applications of the functions $\mathsf{f3},\mathsf{f2}$ and $\mathsf{f1}$.

$$\begin{array}{rl} e = & \mathsf{let}\ \ \mathsf{twice} = \ \mathsf{fn}^{l_1}\,\mathsf{f} \Rightarrow \mathsf{fn}^{l_2}\,\mathsf{x} \Rightarrow \mathsf{f}(\mathsf{f}(\mathsf{x})) \\ & \mathsf{in}\ \ \mathsf{let}\ \ \mathsf{f1} = \mathsf{fn}^{l_3}\,\mathsf{x} \Rightarrow () \\ & \qquad \mathsf{in}\ \mathsf{let}\ \ \mathsf{f2} = \ \mathsf{fn}^{l_4}\,\mathsf{x} \Rightarrow \mathsf{twice}\ \ \mathsf{f1}\,\mathsf{x} \\ & \qquad\quad \mathsf{in}\ \mathsf{let}\ \ \mathsf{f3}\ = \mathsf{fn}^{l_5}\,\mathsf{x} \Rightarrow \mathsf{twice}\ \ \mathsf{f2}\,\mathsf{x} \\ & \qquad\qquad \mathsf{in}\ \mathsf{twice}\ \ \mathsf{f3}\ () \end{array}$$

4.3 A Monomorphic type system

It would be possible to obtain a monomorphic type system for rEval-λ simply by extending the basic types of the Core language with a type for site names. However, we do not only want to design a type system which enjoys a soundness property but also to be able to exploit types for distributed-call tracking.

4.3.1 Semantic objects

The idea of enriching types with annotations for static estimation purposes has already appeared in Chapter 3. The type systems of this chapter build on similar intuitions. The values which are of interest to the problem in hand are determined and expressions which yield these kinds of values are labelled. The type system then examines a program collecting information about its possible behaviours. This is achieved through incorporating labels into types as annotations.

The values of interest for distributed-call tracking analysis are functions and sites. Therefore, we annotate function and site types only. If an expression has a site type in our system, the annotation of its type estimates the site names which can result from evaluating this expression. If an expression has a function type, the annotation of its type estimates the functions which may be called during its evaluation.

Site annotations are sets of site names. Flow annotations are sets whose elements can be function labels or annotations of the form $\{\mathcal{F} @ \mathcal{S}\}$. A function label stands for a function which is called at the current site and an annotation of the form $\{\mathcal{F} @ \mathcal{S}\}$ is used to represent an estimate of the functions which may be called at the sites in $\mathcal{S}$.

Site annotations	$\mathcal{S}$	::=	$\emptyset \mid \{s\} \mid \mathcal{S}_1 \cup \mathcal{S}_2$
Flow annotations	$\mathcal{F}$	::=	$\emptyset \mid \{l\} \mid \{\mathcal{F} @ \mathcal{S}\} \mid \mathcal{F}_1 \cup \mathcal{F}_2$
Types	τ	::=	$\mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{site}^{\mathcal{S}} \mid \tau_1 \xrightarrow{\mathcal{F}} \tau_2$
Type environments	Γ	::=	$[\,] \mid \Gamma[x \mapsto \tau]$

The only other objects used by our type system are type environments which are defined as finite maps from identifiers to types. We assume that the types of the ubiquitous values are present in the initial type environment.

4.3.2 Typing rules

Our type system for rEval-λ assigns a type (τ) and a flow annotation $(\mathcal{F})$ to each expression. A judgement of the form $\Gamma \vdash e : \tau, \mathcal{F}$ states that an expression has type τ and induces a flow represented by $\mathcal{F}$ where Γ contains the assumptions about the types of free identifers of e. The judgements of this type system can be compared to the judgements of the form $\Gamma \vdash e : \tau, \kappa$ from Chapter 3. In that system we estimated the possible communication effects κ which may be incurred by an expression in much the same way as we are estimating the possible function calls $\mathcal{F}$ in the current type system.

The typing rules presented in Figure 4.4 are mostly self-explanatory. The rule (site) for typing site names ensures that the annotation of the type of a site name includes the site name itself. This provides the means for tracing site names in the static semantics.

The typing rule (fn) requires that the label of a function is included in its type as an element of its flow annotation along with the flow annotation derived for the body of the function. By inspecting the rule (app) for function applications we can see how these annotations are exploited. When a function is applied as a part of the evaluation of an expression, it is guaranteed that the flow annotation derived for the expression will include the label of the applied function.

While presenting the dynamic semantics we stated that a remote evaluation can take place only if the first expression evaluates to a function. The typing rule (reval) ensures that this condition is met. Since the value of the second expression is used as an argument to the function during its remote evaluation, the typing rule also requires that its type is identical to the argument type of the function. The value of the third expression is used to specify the destination for evaluation. Therefore, the typing rule forces its type to be a site type. We capture the fact that the function is to be evaluated at the specified site by a flow annotation of the form $\{\mathcal{F}@\mathcal{S}\}$, which is composed of the annotation of the function type and the site type. The typing of spawn expressions follows the same principle.

Allowing flow annotations of the form $\{\mathcal{F}@\mathcal{S}\}$ makes the typing rules relatively easy to understand. However, it also gives rise to large annotations which are nested under the combinator $@$. We define an operation which converts an arbitrary flow annotation into a set where each element is either of the form l or $l@\mathcal{S}$. Such an operation can be of interest as a method for producing simpler annotations if the flow annotations are to be read by users. It can also be employed by a type inference procedure internally to judge the equivalence of flow annotations which may have different representations.

Definition 4.1 ($\overline{\mathcal{F}}$).

$$\overline{\mathcal{F}} = \begin{cases} \emptyset & \text{if } \mathcal{F} = \emptyset \text{ or } \mathcal{F} = \emptyset @ \mathcal{S} \\ \{l\} & \text{if } \mathcal{F} = \{l\} \\ \{l @ \mathcal{S}\} & \text{if } \mathcal{F} = \{l\} @ \mathcal{S} \\ \overline{\{\mathcal{F}' @ \mathcal{S}'\}} & \text{if } \mathcal{F} = \{(\mathcal{F}' @ \mathcal{S}') @ \mathcal{S}\} \\ \overline{\{\mathcal{F}' @ \mathcal{S}\}} \cup \overline{\{\mathcal{F}'' @ \mathcal{S}\}} & \text{if } \mathcal{F} = \{(\mathcal{F}' \cup \mathcal{F}'') @ \mathcal{S}\} \\ \overline{\mathcal{F}'} \cup \overline{\mathcal{F}''} & \text{if } \mathcal{F} = \mathcal{F}' \cup \mathcal{F}'' \end{cases}$$

When there are multiple layers of site annotations enclosing a label, it is the innermost layer which determines the site of the function call. For example, an annotation such as $\{l_0, l_1 @ \{s_1\}\} @ \{s_2\}$ expresses that the annotation $\{l_0, l_1 @ \{s_1\}\}$ estimates the control flow at s_2. The function with label l_0 may be called at site s_2, and the evaluation at site s_2 may trigger the remote evaluation of the function with label l_1 at site s_1. The operation defined above can be used to perform this simplification. It can be checked that $\overline{\{l_0, l_1 @ \{s_1\}\} @ \{s_2\}} = \{l_0 @ \{s_2\}, l_1 @ \{s_1\}\}$.

The typing rule (subs) allows a flow annotation $\mathcal{F}$ to be replaced by another one $\mathcal{F}'$ if the set obtained by simplifying $\mathcal{F}'$ is at least as large as the simplified form of $\mathcal{F}$.

4.3.3 Examples

We now discuss which types can be assigned by our type system to the examples of Section 4.2.3.

Revisiting Example 4.1 Let us assume that the function $\mathsf{getTime}$ is a ubiquitous function and the typing takes place with respect to the initial type environment Γ where $\Gamma(\mathsf{getTime}) = \mathsf{unit} \xrightarrow{\{l_g\}} \mathsf{int}$. The following types can be derived for the function $\mathsf{remoteTime}$ and the expression e.

- $\Gamma \vdash \mathsf{remoteTime} : \mathsf{site}^{\{s1\}} \xrightarrow{\{l, l_g @ \{s1\}\}} \mathsf{int}, \emptyset$
- $\Gamma \vdash e : \mathsf{int}, \{l, \{l_g\} @ \{s1\}\}$

(con) $\Gamma \vdash () : \mathsf{unit}, \emptyset \qquad \Gamma \vdash n : \mathsf{int}, \emptyset$

$\Gamma \vdash \mathsf{true} : \mathsf{bool}, \emptyset \qquad \Gamma \vdash \mathsf{false} : \mathsf{bool}, \emptyset$

(site) $\Gamma \vdash s : \mathsf{site}^{\mathcal{S}}, \emptyset \qquad \text{where } s \in \mathcal{S}$

(id) $\Gamma \vdash x : \Gamma(x), \emptyset$

(fn) $$\frac{\Gamma[x \mapsto \tau] \vdash e : \tau', \mathcal{F}}{\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e : \tau \xrightarrow{\{l\} \cup \mathcal{F}} \tau', \emptyset}$$

(rec) $$\frac{\Gamma[x \mapsto \tau][f \mapsto \tau \xrightarrow{\{l\} \cup \mathcal{F}} \tau'] \vdash e : \tau', \{l\} \cup \mathcal{F}}{\Gamma \vdash \mathsf{rec}^l\, f\,(x) \Rightarrow e : \tau \xrightarrow{\{l\} \cup \mathcal{F}} \tau', \emptyset}$$

(app) $$\frac{\Gamma \vdash e_1 : \tau \xrightarrow{\mathcal{F}} \tau', \mathcal{F}' \qquad \Gamma \vdash e_2 : \tau, \mathcal{F}''}{\Gamma \vdash e_1\, e_2 : \tau', \mathcal{F} \cup \mathcal{F}' \cup \mathcal{F}''}$$

(op) $$\frac{\Gamma \vdash e_1 : \tau, \mathcal{F} \qquad \Gamma \vdash e_2 : \tau, \mathcal{F}' \qquad op : (\tau * \tau) \xrightarrow{\emptyset} \tau'}{\Gamma \vdash e_1\, op\, e_2 : \tau', \mathcal{F} \cup \mathcal{F}'}$$

(if) $$\frac{\Gamma \vdash e_1 : \mathsf{bool}, \mathcal{F} \qquad \Gamma \vdash e_2 : \tau, \mathcal{F}' \qquad \Gamma \vdash e_3 : \tau, \mathcal{F}''}{\Gamma \vdash \mathsf{if}\ e_1 \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau, \mathcal{F} \cup \mathcal{F}' \cup \mathcal{F}''}$$

(let) $$\frac{\Gamma \vdash e_1 : \tau, \mathcal{F} \qquad \Gamma[x \mapsto \tau] \vdash e_2 : \tau', \mathcal{F}'}{\Gamma \vdash \mathsf{let} x = e_1\, \mathsf{in} e_2 : \tau', \mathcal{F} \cup \mathcal{F}'}$$

(reval) $$\frac{\Gamma \vdash e_1 : \tau \xrightarrow{\mathcal{F}} \tau', \mathcal{F}' \qquad \Gamma \vdash e_2 : \tau, \mathcal{F}'' \qquad \Gamma \vdash e_3 : \mathsf{site}^{\mathcal{S}}, \mathcal{F}'''}{\Gamma \vdash \mathsf{reval}\,(e_1, e_2)\, \mathsf{at} e_3 : \tau', \mathcal{F}' \cup \mathcal{F}'' \cup \mathcal{F}''' \cup \{\mathcal{F} @ \mathcal{S}\}}$$

(spawn) $$\frac{\Gamma \vdash e_1 : \tau \xrightarrow{\mathcal{F}} \tau', \mathcal{F}' \qquad \Gamma \vdash e_2 : \tau, \mathcal{F}'' \qquad \Gamma \vdash e_3 : \mathsf{site}^{\mathcal{S}}, \mathcal{F}'''}{\Gamma \vdash \mathsf{spawn}\,(e_1, e_2)\, \mathsf{at} e_3 : \mathsf{unit}, \mathcal{F}' \cup \mathcal{F}'' \cup \mathcal{F}''' \cup \{\mathcal{F} @ \mathcal{S}\}}$$

(subs) $$\frac{\Gamma \vdash e : \tau, \mathcal{F} \qquad \overline{\mathcal{F}} \subseteq \overline{\mathcal{F}'}}{\Gamma \vdash e : \tau, \mathcal{F}'}$$

Figure 4.4: Monomorphic Typing Rules for rEval-λ

The flow annotation derived for e exposes that the evaluation of e can cause the function with label l to be called at the current site and the function with label l_g, that is function getTime, to be called at site s1.

Revisiting Example 4.2 Let us assume that thisHost is a ubiquitous function and $\Gamma(\mathsf{thisHost}) = \mathsf{unit} \xrightarrow{\{l_t\}} \mathsf{site}^{\{s0,s1\}}$. The following types can be derived for the recursive function f and the expression e.

- $\Gamma \vdash \mathsf{f} : \mathsf{site}^{\{s0,s1\}} \xrightarrow{\{l,l_t,\{l,l_t\}@\{s0,s1\}\}} \mathsf{unit}, \emptyset$
- $\Gamma \vdash e : \mathsf{unit}, \{l, l_t, \{l, l_t\}@\{s0, s1\}\}$

The argument type of function f reveals that the actual arguments to function f at run-time may be either s0 or s1. The flow annotation derived for e indicates that functions with label l and l_t can be called at the site where the type-checking has taken place and also at a site estimated by the set $\{s0, s1\}$. The fact that the evaluation of a recursive function may span two sites could be considered as potentially dangerous for certain systems.

Revisiting Example 4.3 The following types can be derived for the expression e and the functions embodied by it where $\mathcal{F} = \{l_1, l_2, l_3, l_4, l_5\}$. The initial type environment has no bearing upon the typing in this case.

- $\Gamma \vdash \mathsf{twice} : (\mathsf{unit} \xrightarrow{\mathcal{F}} \mathsf{unit}) \xrightarrow{\{l_1\}} \mathsf{unit} \xrightarrow{\mathcal{F}} \mathsf{unit}, \emptyset$
- $\Gamma' \vdash \mathsf{f1} : (\mathsf{unit} \xrightarrow{\mathcal{F}} \mathsf{unit}), \emptyset$
  where $\Gamma' = \Gamma[\mathsf{twice} \mapsto (\mathsf{unit} \xrightarrow{\mathcal{F}} \mathsf{unit}) \xrightarrow{\{l_1\}} \mathsf{unit} \xrightarrow{\mathcal{F}} \mathsf{unit}]$
- $\Gamma'' \vdash \mathsf{f2} : (\mathsf{unit} \xrightarrow{\mathcal{F}} \mathsf{unit}), \mathcal{F}$
  where $\Gamma'' = \Gamma'[\mathsf{f1} \mapsto (\mathsf{unit} \xrightarrow{\mathcal{F}} \mathsf{unit})]$
- $\Gamma''' \vdash \mathsf{f3} : (\mathsf{unit} \xrightarrow{\mathcal{F}} \mathsf{unit}), \mathcal{F}$
  where $\Gamma''' = \Gamma''[\mathsf{f2} \mapsto (\mathsf{unit} \xrightarrow{\mathcal{F}} \mathsf{unit})]$
- $\Gamma \vdash e : \mathsf{unit}, \mathcal{F}$

The flow annotation derived for e rightly expresses that the evaluation of e would cause all the functions in the code to be invoked. The types of the functions $\mathsf{f1}, \mathsf{f2}, \mathsf{f3}$ are identical because they are all passed as an argument to the function twice. If a higher-degree of precision is required then a modification of the type system becomes necessary. In Section 4.4 we propose a more complex type system which improves upon the monomorphic type system in terms of expressiveness and precision.

4.3.4 Formal properties

One of the properties we prove about our type system is its consistency with the dynamic semantics. This involves showing that types are preserved under transitions of the system and that flow annotations are consistent with the annotations of the dynamic semantics.

4.3.4.1 Types for blocked expressions

Showing a type preservation property requires us to be able to derive types for all forms of intermediate expressions. We introduce the following typing rule for blocked expressions in addition to the typing rules of Figure 4.4.

$$\frac{\Gamma \vdash e : \tau, \mathcal{F} \quad \mathcal{F} \subseteq \mathcal{F}'}{\Gamma \vdash \mathsf{block}_{(s,p)}\,\mathsf{on}\, e : \tau, \{\mathcal{F}'@s\}}$$

The flow annotation $\{\mathcal{F}'@s\}$ reflects the fact that the calls incurred by the evaluation of e take place at s. The evaluation of e must have been triggered by a function call in the first place. This is accommodated by allowing $\mathcal{F}'$ to be larger than $\mathcal{F}$.

4.3.4.2 Consistency

The annotations on the dynamic evaluation rules represent function calls. The flow annotations used in the type system are meant to be static estimates of these actions. The definition below formalizes the static estimation relation between actions (a) and flow annotations ($\mathcal{F}$). We read $a \models \mathcal{F}$ as $\mathcal{F}$ estimates a.

Definition 4.2 **(**$a \models \mathcal{F}$**).**

$$\begin{array}{ll} \varepsilon \models \emptyset & \\ l@s \models \{l@\mathcal{S}\} & \text{if } s \in \mathcal{S} \\ a \models \{l@\mathcal{S}\} \cup A & \text{if } a \models \{l@\mathcal{S}\} \text{ or } a \models A \end{array}$$

If a well-typed expression is reduced to another expression in a single step, the resulting expression must also be an expression of the same type. The first part of the consistency theorem below proves that this is indeed the case. This property is a precondition for ensuring that the evaluation of an expression in multiple steps yields a value of the expected type.

We also need to show that the function call behaviour incurred by the evaluation at run-time is estimated by the flow annotation assigned to the expression in the sense defined above. If a well-typed expression e reduces to another expression e' by possibly calling a function represented by a, the flow annotation of e must estimate a. This is proved by the second part of the consistency theorem. We appeal to the lemma presented below in our proof.

Lemma 4.1 (Expression substitution). If $\Gamma[x \mapsto \tau] \vdash e : \tau', \mathcal{F}$ and $\Gamma \vdash e' : \tau, \emptyset$. Then $\Gamma \vdash e\{e'/x\} : \tau', \mathcal{F}$.

Proof. The proof is by induction on the depth of the typing derivation. □

Theorem 4.1 (Consistency). Let e be a closed expression which is evaluated at site s in the system. Assume $S[(s,p) : e] \xrightarrow{a} S'[(s,p) : e']$ and $\Gamma \vdash e : \tau, \mathcal{F}$. Then $\Gamma \vdash e' : \tau, \mathcal{F}$ and $a \models \overline{\{\mathcal{F}@\{s\}\}}$.

Proof. The proof is given by induction on the depth of the derivation of $S[(s,p) : e] \xrightarrow{a} S'[(s,p) : e']$. It refers to Lemma 4.1. Two selected cases are given in the Appendix. Note that we can assume that the non-structural (subs) rule is used after every structural rule and nowhere else. Any derivation tree for $\Gamma \vdash e : \tau, \mathcal{F}$ can be transformed into a derivation tree where we use the rule (subs) after every structural rule. By transitivity of the subset relation $\subseteq$ on sets we can eliminate multiple applications of the (subs) rule and consider it as a single application of the rule. □

4.3.4.3 Minimal types

Given an expression e and an initial type environment Γ, there may be zero, one or multiple typing derivations which conform to the typing rules of our system. This is mainly due to the possibility of deriving annotations which are larger than strictly necessary. Our measure of size for a flow annotation is the size of the set obtained by flattening that annotation with respect to Definition 4.3 below. The precision of estimation increases as the size of annotations decreases. Therefore, the notion of *best type* coincides with the notion of *minimal type*, which is the type with the smallest annotation.

Definition 4.3 ($Flatten$). The operation *Flatten* is defined on flow annotations in which all elements are of the form l or $l@\mathcal{S}$ where $\mathcal{S}$ is non-empty. It yields a flow annotation where all elements are of the form l or $l@s$.

$$Flatten(\mathcal{F}) = \begin{cases} \emptyset & \text{if} \mathcal{F} = \{\emptyset@\mathcal{S}\} \text{ or } \mathcal{F} = \emptyset \\ \{l\} & \text{if } \mathcal{F} = \{l\} \\ \{l@s\} & \text{if } \mathcal{F} = \{l@\{s\}\} \\ \{l@s\} \cup Flatten(\{l@\mathcal{S}\}) & \text{if } \mathcal{F} = \{l@(\{s\} \cup \mathcal{S})\} \end{cases}$$

Note that the operation of Definition 4.1 transforms an arbitrary $\mathcal{F}$ into the form on which the operation *Flatten* is defined.

Definition 4.4 (Ordering $\sqsubseteq$ on types).

basic types	$\tau \sqsubseteq \tau$	if τ is $\mathsf{int}, \mathsf{unit}$ or bool
sites	$\mathsf{site}^{\mathcal{S}} \sqsubseteq \mathsf{site}^{\mathcal{S}'}$	if $\mathcal{S} \subseteq \mathcal{S}'$
functions	$\tau_1 \xrightarrow{\mathcal{F}} \tau_2 \sqsubseteq \tau_1' \xrightarrow{\mathcal{F}'} \tau_2'$	if $\tau_1 \sqsubseteq \tau_1'$ and $\tau_2 \sqsubseteq \tau_2'$ and $\mathcal{F} \sqsubseteq \mathcal{F}'$
flow annotations	$\mathcal{F} \sqsubseteq \mathcal{F}'$	if $Flatten(\mathcal{F}) \subseteq Flatten(\mathcal{F}')$

We note that the ordering $\sqsubseteq$ can be extended to type environments in a pointwise manner with $[] \sqsubseteq \Gamma$.

Definition 4.5 (Annotation erasure ($\lfloor . \rfloor$)). The operation $\lfloor . \rfloor$ is defined on annotated types. It erases all the annotations on the types and yields a simple (non-annotated) type. The definition of $\lfloor . \rfloor$ extends to type environments such that $\lfloor \Gamma \rfloor(x) = \lfloor \Gamma(x) \rfloor$ for every x in the domain of Γ.

Proposition 4.1 (Minimal types). Let I be a non-empty set of indices and J be the set of possible typing judgements for an expression e defined as follows:

$$J = \{\Gamma^i \vdash e : \tau^i, \mathcal{F}^i \mid i \in I, \lfloor \Gamma^j \rfloor = \lfloor \Gamma^k \rfloor, \lfloor \tau^j \rfloor = \lfloor \tau^k \rfloor \text{ for all pairs } j,k \in I\}$$

Then there exists a minimal element of J, written as $\sqcap\Gamma \vdash e : \sqcap\tau, \sqcap\mathcal{F}$, such that for all $i \in I$ it is the case that $\sqcap\Gamma \sqsubseteq \Gamma^i$ and $\sqcap\tau \sqsubseteq \tau^i$ and $\sqcap\mathcal{F} \sqsubseteq \mathcal{F}^i$.

Proof. The proof is given by induction on the depth of the typing derivation for e. We present the proof for the selected cases in the Appendix. The proof method is an adaptation of that presented in [NNH99b] which shows the existence of minimal solutions for a type-based control flow analysis for a sequential functional language. The authors present two type systems for the language which use non-annotated types and annotated types respectively. They show that whenever a non-annotated type exists for an expression, there exists a flow-annotated type and the possible flow-annotated typing judgements form a Moore family. We remain close in spirit to their approach but establish the truth of our proposition without formally introducing the concepts of lattices and Moore families. Our proposition is formulated for a non-empty set of flow-annotated judgements. However, we note that if an expression is well-typed with respect to a conventional type system which uses non-annotated types, a flow-annotated type can also be derived. This can be achieved by annotating the site types with the set of all site names which appear in the expression. Further, one annotates function types with a union of the set of all function labels and a flow annotation which combines with @ the set of all function labels and the set of all site names.

$\square$

4.4 A Polymorphic type system

In this section we present a type system which can be distinguished from the type system of the previous chapter in two main respects. Firstly, it allows types to be parametric in their flow annotations. Secondly, the form of flow annotations are modified to be able to expose the multiplicity of function calls.

We assume familiarity with the basic concepts and definitions regarding polymorphic type systems as we have already presented one such type system in Chapter 3. The definitions of type substitution and type generalization can be adapted to the type system of this section by replacing communication effects with flow annotations.

4.4.1 Semantic objects

In our polymorphic type system we continue to use sets as site annotations. The crucial extension is that it is now possible to have site types with variable annotations. The meta-variable ρ ranges over site name variables.

A flow annotation of the form $*$ is used to denote the absence of a function call. Function labels and annotations of the form $\mathcal{F}\,@\,\mathcal{S}$ are used in a similar fashion to their counterparts in the monomorphic type system. Parametricity of flow annotations is supported though the incorporation of flow annotation variables into types. The meta-variable ϕ ranges over flow annotation variables.

An annotation of the form $\mathsf{rec}^l\,\phi.\mathcal{F}$ represents a call to a recursive function with label l that exhibits the function call behaviour $\mathcal{F}$. The recursive nature of the call is captured by the fact that the variable ϕ may appear free in $\mathcal{F}$. The combinator $\triangleright$ is introduced to express the sequencing of function calls.

Site annotations	$\mathcal{S}$	::=	$\emptyset \mid \{s\} \mid \rho \mid \mathcal{S}_1 \cup \mathcal{S}_2$
Flow annotations	$\mathcal{F}$	::=	$* \mid l \mid \mathcal{F}\,@\,\mathcal{S} \mid \phi \mid \mathsf{rec}^l\,\phi.\mathcal{F} \mid \mathcal{F}_1 \triangleright \mathcal{F}_2$
Types	τ	::=	$\mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{site}^{\mathcal{S}} \mid \tau_1 \xrightarrow{\mathcal{F}} \tau_2$
Type Schemes	σ	::=	$\forall\vec{\rho}\vec{\phi}.\tau$
Type environments	Γ	::=	$[\,] \mid \Gamma[x \mapsto \sigma]$

We define an equivalence on flow annotations which allows us to simplify flow annotations by using equivalences where possible.

4.4.2 Typing rules

The typing rules for the polymorphic type system are presented in Figure 4.5. They are similar to those of the monomorphic type system. We replace the empty flow annotations ($\emptyset$) with the annotation ($*$) and set union ($\cup$) with sequencing ($\triangleright$). Replacing set union with sequencing improves upon the monomorphic type system in that it allows us to trace how many times a function is called. For example, the set $\{l\} \cup \{l\}$ which is the same as the set $\{l\}$ can only express the possibility of a call to the function with label l. This can be contrasted with the annotation $l \triangleright l$ which expresses that l may be called twice.

The typing rules for constants and identifiers are straightforward. The typing rule (fn) for functions appends to the label of the function the flow annotation derived from the function body. This enables the type to expose that it is the invocation of the function with label l which triggers the subsequent function calls estimated by $\mathcal{F}$. Recursive expressions are assigned types with recursive flow annotations by the rule (rec).

Conditional expressions are typed with respect to the rule (if). The type of a conditional expression is the same as the type of its branches. However, its flow annotation is the combination of those of its subexpressions with $\triangleright$. This gives a rather crude approximation of the actual function call behaviour. A more sophisticated type system such as that of [NN94] could be designed by allowing flow annotations to represent choice via the combinator $+$. In that case we would assign the flow annotation $\mathcal{F} \triangleright (\mathcal{F}' + \mathcal{F}'')$ to the conditional expression. We have, however, chosen to keep the types as simple as possible at the expense of precision. Our main concern in this section is to introduce an approach of incorporating polymorphism to the monomorphic type system of the previous section.

The (let) rule is where the type generalization occurs in a typing derivation. The annotation variables which appear in a type are generalized only if they do not appear free in the typing environment and the flow annotation derived for the expression.

The typing rule (equiv) is an analogue of the typing rule (subs) of the monomorphic type system.

Definition 4.6 (Equivalence of Annotations $\equiv$).

$$
\begin{aligned}
&(*@\mathcal{S}) \equiv * \\
&(* \rhd \mathcal{F}) \equiv \mathcal{F} \\
&(\mathcal{F} \rhd *) \equiv \mathcal{F} \\
&\mathsf{rec}^l\,\phi.\mathcal{F} \equiv \mathcal{F}[\mathsf{rec}^l\,\phi.\mathcal{F}/\phi]
\end{aligned}
$$

4.4.3 Examples

In Section 4.3.3, after presenting the rules of our monomorphic type system, we discussed which types can be assigned to the examples of Section 4.2.3. We now proceed in a similar fashion and present possible typings for the same examples.

Revisiting Example 4.1 Our assumption about the ubiquitous function getTime remains the same; $\Gamma(\mathsf{getTime}) = \mathsf{unit}\xrightarrow{l_g}\mathsf{int}$. In the polymorphic type system it is possible to derive the following types for the function remoteTime and the expression e.

- $\Gamma \vdash \mathsf{remoteTime} : \forall\rho.\mathsf{site}^\rho \xrightarrow{l \rhd (l_g @ \rho)} \mathsf{int}, *$
- $\Gamma \vdash e : \mathsf{int}, l \rhd (l_g @ s1)$

The type of the function remoteTime is parametric in site names. It reveals that the function remoteSite expects a site name as an argument and invokes the function getTime at the site which is passed as an argument to it. In our expression e this site happens to be s1 and the variable ρ gets instantiated to $s1$ in the flow annotation of e. However, we note that the function remoteTime could be used polymorphically. Another expression in the scope of remoteTime could call it with a different site name.

Revisiting Example 4.2 Let us assume that Γ is the initial type environment at s0 and thisHost is a ubiquitous function such that $\Gamma(\mathsf{thisHost}) = \forall\rho.\mathsf{unit}\xrightarrow{l_t}\mathsf{site}^{\{s0,\rho\}}$. It is possible to derive the following types for the recursive function f and the expression e.

(con) $$\Gamma \vdash () : \mathsf{unit}, * \qquad \Gamma \vdash n : \mathsf{int}, *$$

$$\Gamma \vdash \mathsf{true} : \mathsf{bool}, * \quad \Gamma \vdash \mathsf{false} : \mathsf{bool}, *$$

(site) $$\Gamma \vdash s : \mathsf{site}^{\mathcal{S}}, * \quad \text{where } s \in \mathcal{S}$$

(id) $$\frac{\Gamma(x) = \sigma \quad \sigma \succ \tau}{\Gamma \vdash x : \tau, *}$$

(fn) $$\frac{\Gamma[x \mapsto \tau] \vdash e : \tau', \mathcal{F}}{\Gamma \vdash \mathsf{fn}^{l}\, x \Rightarrow e : \tau \xrightarrow{l \rhd \mathcal{F}} \tau', *}$$

(rec) $$\frac{\Gamma[x \mapsto \tau][f \mapsto \tau \xrightarrow{\mathsf{rec}^{l}\phi.\mathcal{F}} \tau'] \vdash e : \tau', \mathsf{rec}^{l}\phi.\mathcal{F}}{\Gamma \vdash \mathsf{rec}^{l}\, f\,(x) \Rightarrow e : \tau \xrightarrow{\mathsf{rec}^{l}\phi.\mathcal{F}} \tau', *}$$

(app) $$\frac{\Gamma \vdash e_1 : \tau \xrightarrow{\mathcal{F}} \tau', \mathcal{F}' \quad \Gamma \vdash e_2 : \tau, \mathcal{F}''}{\Gamma \vdash e_1\, e_2 : \tau', \mathcal{F}' \rhd \mathcal{F}'' \rhd \mathcal{F}}$$

(op) $$\frac{\Gamma \vdash e_1 : \tau, \mathcal{F} \quad \Gamma \vdash e_2 : \tau, \mathcal{F}' \quad op : (\tau * \tau) \xrightarrow{*} \tau'}{\Gamma \vdash e_1\; op\; e_2 : \tau', \mathcal{F} \rhd \mathcal{F}'}$$

(if) $$\frac{\Gamma \vdash e_1 : \mathsf{bool}, \mathcal{F} \quad \Gamma \vdash e_2 : \tau, \mathcal{F}' \quad \Gamma \vdash e_3 : \tau, \mathcal{F}''}{\Gamma \vdash \mathsf{if}\ e_1 \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau, \mathcal{F} \rhd \mathcal{F}' \rhd \mathcal{F}''}$$

(let) $$\frac{\Gamma \vdash e_1 : \tau, \mathcal{F} \quad \Gamma[x \mapsto Gen(\Gamma, \mathcal{F}, \tau)] \vdash e_2 : \tau', \mathcal{F}'}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau', \mathcal{F} \rhd \mathcal{F}'}$$

(reval) $$\frac{\Gamma \vdash e_1 : \tau \xrightarrow{\mathcal{F}} \tau', \mathcal{F}' \quad \Gamma \vdash e_2 : \tau, \mathcal{F}'' \quad \Gamma \vdash e_3 : \mathsf{site}^{\mathcal{S}}, \mathcal{F}'''}{\Gamma \vdash \mathsf{reval}\,(e_1, e_2)\, \mathsf{at}\, e_3 : \tau', \mathcal{F}' \rhd \mathcal{F}'' \rhd \mathcal{F}''' \rhd \mathcal{F}\, @\, \mathcal{S}}$$

(spawn) $$\frac{\Gamma \vdash e_1 : \tau \xrightarrow{\mathcal{F}} \tau', \mathcal{F}' \quad \Gamma \vdash e_2 : \tau, \mathcal{F}'' \quad \Gamma \vdash e_3 : \mathsf{site}^{\mathcal{S}}, \mathcal{F}'''}{\Gamma \vdash \mathsf{spawn}\,(e_1, e_2)\, \mathsf{at}\, e_3 : \mathsf{unit}, \mathcal{F}' \rhd \mathcal{F}'' \rhd \mathcal{F}''' \rhd \mathcal{F}\, @\, \mathcal{S}}$$

(equiv) $$\frac{\Gamma \vdash e : \tau, \mathcal{F} \quad \mathcal{F} \equiv \mathcal{F}'}{\Gamma \vdash e : \tau, \mathcal{F}'}$$

Figure 4.5: Polymorphic Typing Rules for rEval-λ

- $\Gamma \vdash \mathsf{f} : \forall \rho.\mathsf{site}^{\{s0,\rho\}} \xrightarrow{\mathsf{rec}^l\, \phi.l_t \triangleright (\phi @ \{s0,\rho\})} \mathsf{unit}$
- $\Gamma \vdash e : \mathsf{unit}, \mathsf{rec}^l\, \phi.l_t \triangleright (\phi @ \{s0, s1\})$

The type of the recursive function f reveals that f expects a site name as an argument. The annotation of the site type is requested to be $\{s0,\rho\}$ in the typing derivation. The flow annotation shows the recursive nature of the function f. After calling the function with label l_t at the current site, it calls itself at a site estimated by the set $\{s0,\rho\}$. The presence of the variable ρ in both the argument type and the type of the destination for the recursive call may hint at a potentially non-terminating computation.

At first sight it may not be clear why we assign the type $\mathsf{site}^{\{s0,\rho\}}$ to the parameter of f rather than just ρ. The reason is to do with the fact that we do not allow functions to be used polymorphically within their own definition. The type of the argument is required to be identical to the type of the result of the call to $\mathsf{thisHost}$ and we know that any instantiation of the result type of $\mathsf{thisHost}$ will have $s0$ as a part of its annotation.

Revisiting Example 4.3 In Section 4.3.3, we observed that the types of all of the functions which were passed as an argument to the function twice had to be identical. In the current type system this is no longer necessary due to the possibility of giving a polymorphic type to the function twice. The following are possible typings for functions $\mathsf{twice}, \mathsf{f1}, \mathsf{f2}$ and $\mathsf{f3}$.

- $\Gamma \vdash \mathsf{twice} : \forall \phi.(\mathsf{unit} \xrightarrow{\phi} \mathsf{unit}) \xrightarrow{l_1} \mathsf{unit} \xrightarrow{l_2 \triangleright \phi \triangleright \phi} \mathsf{unit}, *$
- $\Gamma' \vdash \mathsf{f1} : (\mathsf{unit} \xrightarrow{\mathcal{F}'} \mathsf{unit}), *$
  where $\mathcal{F}' = l_3$ and $\Gamma' = \Gamma[\mathsf{twice} \mapsto \forall \phi.(\mathsf{unit} \xrightarrow{\phi} \mathsf{unit}) \xrightarrow{l_1} \mathsf{unit} \xrightarrow{l_2 \triangleright \phi \triangleright \phi} \mathsf{unit}]$
- $\Gamma'' \vdash \mathsf{f2} : (\mathsf{unit} \xrightarrow{\mathcal{F}''} \mathsf{unit}), *$
  where $\mathcal{F}'' = l_4 \triangleright l_1 \triangleright l_2 \triangleright \mathcal{F}' \triangleright \mathcal{F}'$ and $\Gamma'' = \Gamma'[\mathsf{f1} \mapsto (\mathsf{unit} \xrightarrow{\mathcal{F}'} \mathsf{unit})]$
- $\Gamma''' \vdash \mathsf{f3} : (\mathsf{unit} \xrightarrow{\mathcal{F}'''} \mathsf{unit}), *$
  where $\mathcal{F}''' = l_5 \triangleright l_1 \triangleright l_2 \triangleright \mathcal{F}'' \triangleright \mathcal{F}''$ and $\Gamma''' = \Gamma''[\mathsf{f2} \mapsto (\mathsf{unit} \xrightarrow{\mathcal{F}''} \mathsf{unit})]$
- $\Gamma \vdash e : \mathsf{unit}, l_1 \triangleright l_2 \triangleright \mathcal{F}''' \triangleright \mathcal{F}'''$

This example also illustrates that imposing a structure on the flow annotations enables us to observe the multiplicity of functions calls. The flow annotation does not only reveal which functions are called but also the number of times that each function is called.

4.4.4 Formal properties

In this section we prove the consistency of the dynamic semantics of rEval-λ and the static semantics which is based on a polymorphic type system. The technical development is similar to that of Section 4.3.4.

4.4.4.1 Types for blocked expressions

$$\frac{\Gamma \vdash e : \tau, \mathcal{F} \quad \mathcal{F}' \equiv l \rhd \mathcal{F} \text{ for some } l}{\Gamma \vdash \mathsf{block}_{(s,p)}\, \mathsf{on}\, e : \tau, \mathcal{F}'@s}$$

This typing rule is based on a similar intuition to the typing rule for blocked expressions in the monomorphic type system. Since expressions of this form arise as a consequence of a function being sent to a remote site for evaluation, the remotely evaluated function's label must appear in $\mathcal{F}'$.

4.4.4.2 Consistency

The relation ($\models_p$) below relates the flow annotations of the dynamic semantics to flow annotations of the static semantics.

Definition 4.7 ($\models_p$). $\varepsilon \models_p *$
$l@\{s\} \models_p l@\mathcal{S} \quad \text{if } s \in \mathcal{S}$
$a \models_p (l@\mathcal{S}) \rhd A \quad \text{if } a \models_p l@\mathcal{S} \text{ or } a \models_p A$

Definition 4.8 ($\overline{\overline{\mathcal{F}}}$). The operation $\overline{\overline{\mathcal{F}}}$ is analogous to the operation $\overline{\mathcal{F}}$ defined for simplifying the flow annotations of the monomorphic type system.

$$\overline{\overline{\mathcal{F}}} = \begin{cases} \emptyset & \text{if } \mathcal{F} = * \text{ or } \mathcal{F} = *@\mathcal{S} \\ l & \text{if } \mathcal{F} = l \\ l@\mathcal{S} & \text{if } \mathcal{F} = l@\mathcal{S} \\ \phi@\mathcal{S} & \text{if } \mathcal{F} = \phi@\mathcal{S} \\ \overline{\overline{\mathcal{F}'}} & \text{if } \mathcal{F} = \mathsf{rec}^l\,\phi.\mathcal{F}' \\ \overline{\overline{\mathcal{F}'@\mathcal{S}'}} & \text{if } \mathcal{F} = (\mathcal{F}'@\mathcal{S}')@\mathcal{S} \\ \overline{\overline{\mathcal{F}'@\mathcal{S}}} \triangleright \overline{\overline{\mathcal{F}''@\mathcal{S}}} & \text{if } \mathcal{F} = \{(\mathcal{F}' \triangleright \mathcal{F}'')@\mathcal{S}\} \\ \overline{\overline{\mathcal{F}'}} \triangleright \overline{\overline{\mathcal{F}''}} & \text{if } \mathcal{F} = \mathcal{F}' \triangleright \mathcal{F}'' \end{cases}$$

Lemma 4.2 (Type substitution). If $\Gamma \vdash e : \tau, \mathcal{F}$ then $\theta\Gamma \vdash e : \theta\tau, \theta\mathcal{F}$ for any substitution θ.

Lemma 4.3 (Expression substitution). If $\Gamma[x \mapsto Gen(\Gamma, *, \tau)] \vdash e : \tau', \mathcal{F}$ and $\Gamma \vdash e' : \tau, *$ then $\Gamma \vdash e\{e'/x\}\tau', \mathcal{F}$.

The consistency theorem is similar to Theorem 4.1. It states that types are preserved under transitions. It is sufficient to show that the flow annotation of the expression produced as a result of one step reduction is a subsequence of the expression which is being reduced. Whatever function call might occur must be estimated by the flow annotation of the expression which goes under reduction.

Theorem 4.2 (Consistency). Let e be a closed expression which is evaluated at site s. Assume $S[(s,p) : e] \xrightarrow{a} S'[(s,p) : e']$ and $\Gamma \vdash e : \tau, \mathcal{F}$. Then $\Gamma \vdash e' : \tau, \mathcal{F}'$ where $\overline{\overline{\mathcal{F}'}}$ is a subsequence of $\overline{\overline{\mathcal{F}}}$ and $a \models_p \overline{\overline{\mathcal{F}@s}}$.

Proof. Two selected cases of the proof are given in the Appendix. □

4.5 Concluding remarks

When we introduced the term distributed-call tracking analysis, we mentioned that the idea of using type systems for estimating which functions are called during the execution of a program is not novel. Type systems for call-tracking analysis have already been investigated within the context of higher-order sequential functional languages [Hei95, TJ92]. The novelty of our work lies in the fact that our type systems account for the presence of different localities in a system. The invocation of a function at one site is distinguished from its invocation at another site.

Types as interface descriptions Our work also draws attention to the important role played by types as concise descriptions of programs. In a functional language such as rEval-λ the type of a function serves as its interface description which tells how it should be used, which other functions it may call and which sites its evaluation may span. In a security-sensitive system such descriptions generated by credible sources would be useful not only for automated tools that rely on this information but for users who wish to use services developed by other parties.

Types and termination analysis We have pointed out the connection between termination of programs and security threats caused by denial of service. This suggests that static termination analyses could be exploited for strengthening the security properties of a language. An approach based on types for termination analysis has been studied in [NN96a]. This work focuses on a higher-order functional language with algebraic datatypes. The main idea is to check by means of types whether the size of arguments to recursive calls decreases as the computation evolves. It requires defining a well-founded partial order for the data types which may be passed as arguments to functions. The applicability of this work in the framework of a language such as rEval-λ would be an interesting subject for future research. However, the work presented in this chapter is not aimed at termination analysis in particular.

Types and resource-bounds We have suggested that one of the application areas of distributed-call tracking could be execution time analysis. This relates our work to the line of related research on using type systems for estimating the time-complexity of programs [DJG92, RG94]. A polymorphic type system which estimates the execution time of expressions in a functional language has been presented in [DJG92]. In this type system function types carry as annotation the estimated number of clock ticks for the execution of a function. The time required for the execution of an expression is derived in a compositional manner, by algebraic manipulations of the estimated time for the execution of its subexpressions. Types of recursive functions have an annotation of a special form which states that the execution may take an arbitrarily long time. The time estimates are suggested to be useful for determining where the code optimization effort should be concentrated and when it is worthwhile to exploit parallelising the execution of a program on multiprocessors. More recently, some authors have pointed out the potential benefits of using type systems for specifying and certifying resource bounds such as bounds on running time [CW00]. A decidable type system for a functional language has been presented in [CW00]. Our work shares its motivation with these works. However, distributed-call tracking can be of interest for reasons other than execution time estimation because it provides information about the identities of functions and their call sites.

Using type systems to guarantee resource bounds on space consumption of functional languages has also been of interest to several authors [Hof00, HP99]. A possible direction of future work could be to extend the language rEval-λ with inductively defined datatypes such as lists and language constructs for manipulating these. It would then be possible to investigate variants of the type systems of this chapter to estimate the space consumption of programs at different sites within a system.

Alternative forms of remote evaluation The support for remote evaluation is a characteristic feature of the language rEval-λ. Expressions can be sent for evaluation at a remote site in the distributed system and the sender blocks until the result is returned back to it by the site which has computed the result. It can be possible to adopt a more flexible remote evaluation model for rEval-λ which is inspired by the mechanism of futures [Hal85]. After initiating a remote evaluation the sender can resume its computation until an attempt is made to use the result of the remotely evaluated expression. We believe that the ideas and techniques presented in this chapter would also be applicable to tackle the problem of distributed-call tracking analysis for such a language.

Chapter 5

Confined Mobile Functions

It is becoming increasingly common for distributed systems to bring together computing devices of different processing power, software provided by different sources and information with different secrecy and integrity requirements. Moreover, the users which interact with the system may be of different trust levels.

The ability to distribute computation among different sites is desirable because it enables effective exploitation of the resources in a system. However, in the absence of appropriate protection mechanisms it may also lead to uncontrolled use of these resources and undesirable information flows. It is important that applications be designed with concern for meeting the security requirements of the system.

The aim of this chapter is to introduce a language mechanism which gives programmers a means to control the distribution of computation and the flow of information throughout the system. We consider a simple programming language – Confined-λ – which allows programmers to declare a *mobility region* for the services and the information they provide. The mobility region of an entity determines the subsystem in which it can flow freely. We propose a static type-system for our language which enforces the property called *confinement in a mobility region*. This property guarantees that the entities created and manipulated by well-typed Confined-λ programs will remain within their specified mobility regions at run-time.

5.1 Why restrict mobility?

Before introducing the language Confined-λ in detail we discuss why a property such as confinement in a mobility region could be of interest to programmers. For the purposes of this section it is sufficient to note that the language Confined-λ resembles Mobile-λ from Chapter 3. It supports channel-based communication where values of all types can be communicated on channels. In an application based on the design paradigm of code on demand [FPV98], services would be implemented as functions and the requesters would access these functions by retrieving them over a channel.

The motivation for controlling the mobility of a basic value would typically be security related. For example, if an integer represents a personal identification number (PIN) for accessing a particular account, its mobility should be restricted to a part of the system which harbours as observers only the authorized users of that account. In a language such as Confined-λ a natural way to realize this would be to associate a group of users with the integer upon its creation.

Functions are different from basic values because they are not passive values which are simply passed around. They abstract a behaviour which becomes activated when the function is applied. The authors may want to constrain the mobility of functions due to performance and security reasons. For example, a function may be using the resources of the system intensively to perform its task. To ensure that the performance of the system is not adversely affected, it would be reasonable to restrict the use of this function to those sites which have sufficient computational resources. Note that such a restriction would also be useful in preventing denial of service within the system. Likewise, it would be desirable to restrict the use of a function which returns private information about a particular user to a part of the system in which all the observers are eligible to obtain this information.

Channels provide the only means of establishing connections between remote sites when computing with Confined-λ. The desired level of connectivity within the system influences the policy on the mobility of channels. On the other hand, channels are instrumental in communicating information within the system. Acquiring a channel implies acquiring the right to input or output on that channel. The policy about the mobility of channels needs to be in accordance with the policy about the mobility of the values it can communicate.

5.2 Computing with mobility regions

5.2.1 System model

Throughout this chapter we consider computation in a multi-site system in which each site has a unique name drawn from a finite set of site names $S = \{s_i \mid 1 \leq i \leq \text{size of the system}\}$. Each site is controlled by a particular user who both provides the code and observes the results of the computation at that site. This implies that a site name also identifies a user; the observation of a value at a particular site or the observation of a value by a particular user are essentially the same concepts. The same holds for the origination of code at a particular site and the origination of code by a particular user.

A program consists of multiple expressions, one for each site participating in the computation. Each expression is type-checked by a trusted Confined-λ compiler and the program is allowed to execute only if all of the expressions are well-typed.

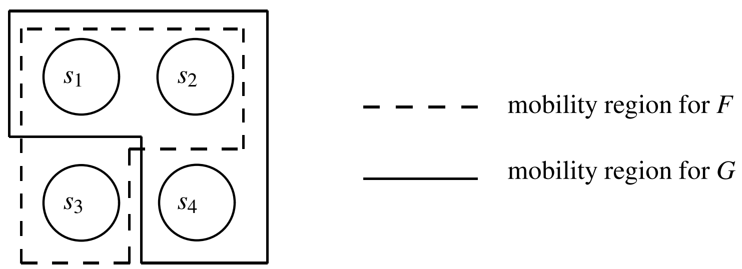
Observables We say that a value v is observable at a given site s, if the computation returns v at site s, or v appears at site s as an intermediate step in the computation. According to this definition, a value v could be observable at site s through being received over a channel at any point in the computation or through being contained in the code of a function which is received at s.

5.2.2 Mobility regions

As we discussed above, our approach is to give users full control over the flow of values which are created by the code written by them. Let us now consider a system which consists of just four sites with the names given in the set $S = \{s_1, s_2, s_3, s_4\}$ and suppose that the user at site s_1 provides the code for a service F which requires extensive processing power and s_4 is known to be equipped with a less powerful processor. It is reasonable for s_1 to restrict the use of F only to s_2 and s_3 besides itself. On the other hand, a service G may be carrying out a simple operation which does not demand much processing power but may return a value which the user at s_1 wants to keep secret from s_3. In this case, G should not be allowed to run at s_3.

We define a mobility region as a non-empty subset of the sites in the system. If we consider the above system with site names drawn from S, any non-empty subset of S is a valid mobility region. From the security perspective, mobility regions are used to declare a web of trust among a group of sites. Information which is required to be confined to a particular mobility region $r = \{s_1, s_2, s_3\}$ may be observed at any site within r but it should not be observed at any site outside r.

Continuing this example, $M_1 = \{s_1, s_2, s_3\}$ and $M_2 = \{s_1, s_2, s_4\}$ are the mobility regions which would be assigned to services F and G respectively. F can visit s_1, s_2, s_3 but not s_4. G can visit s_1, s_2, s_4 but not s_3.



Ordering on mobility regions Mobility regions can be interpreted as secrecy levels where the subset relation on mobility regions gives rise to an ordering on secrecy levels. Suppose that $\sqsubseteq$ is an ordering on secrecy levels where we write $A \sqsubseteq B$ if B indicates a higher secrecy level than A. Then the statements $r_1 \sqsubseteq r_2$ and $r_2 \subseteq r_1$ are equivalent. Note that by using mobility regions it would be possible to express as many secrecy levels as there are non-empty subsets of the site names in a system.

5.3 Confined-λ

The language Confined-λ is similar to Mobile-λ in that it extends the Core language with primitives for communication between remote sites. We omit the conditional expressions because they do not have any significance with regard to the problem considered in this chapter. Their operational meaning and typing would have been similar to those found in the languages of the previous chapters.

5.3.1 Abstract syntax

In Confined-λ canonical expressions and channel allocation expressions are annotated with mobility regions. It is by means of these annotations that programmers specify the constraints on the mobility of values created by their programs.

Site names	s	::=	$s_1 \mid s_2 \mid \ldots$	
Mobility regions	r	::=	$\{s\} \mid r_1 \cup r_2$	
Expressions	e	::=	c^r	annotated constants
			$\mid \mathsf{fn}^r\, x \Rightarrow e$	annotated functions
			$\ldots$	expressions of the Core
			$\mid \mathsf{chan}^r()$	channel allocation
			$\mid e_1!e_2$	send
			$\mid e?$	receive

Figure 5.1: Abstract Syntax for Confined-λ

5.3.2 Dynamic semantics

Canonical expressions, that is the expressions which cannot be further evaluated, denote values of the language Confined-λ.

$$\textbf{Values} \qquad v \;::=\; ()^r \mid n^r \mid k^r \mid \mathsf{fn}^r\, x \Rightarrow e$$

We assume that a channel identifier is represented by a tuple which includes the identifier of the site it is created at and an integer which is freshly generated each time a new channel is allocated at a site.

Definition 5.1 (L_d). Function L_d is defined on values of the language. It yields the label of a given value which denotes its mobility region.

$$L_d(()^r) = r \qquad L_d(n^r) = r \qquad L_d(k^r) = r \qquad L_d(\mathsf{fn}^r\, x \Rightarrow e) = r$$

5.3.2.1 Evaluation rules

The evaluation rules of Figure 5.2 are defined in a similar style to those of the previous chapters. In the rule (op-3) for basic operations v is a basic value obtained by applying operation **op** to v_1 and v_2. We implicitly assume that if $L_d(v_1) = r$ and $L_d(v_2) = r'$ and $v = v_1 \,\mathbf{op}\, v_2$ then $L_d(v) = r \cap r'$.

5.3.3 Examples

In Section 5.2 we introduced mobility regions as a useful device for programmers to specify how they wish to constrain the flow of values. In the next section we will present a set of typing rules, which forces the values to be confined to their specified mobility regions. These typing rules prevent region violations where we use the phrase *region violation* to refer to the observability of a value outside its specified mobility region. Our aim here is to give the intuition behind our typing rules by illustrating major sources of region violations through simple examples.

Example 1: Region violation due to uncontrolled use of channels In the following example, A is the creation site for two values: channel chAC intended for communication between sites A and C and function f which is allowed to roam freely. The function f requires a channel as an argument. It uses this channel to input a value and forwards the input value back to site A by using the channel chAC in its closure. When executed, the expression at A will send f to C on a previously allocated channel chTop which is known to both A and C. It will then start listening on chAC. We use ; here as a shorthand for sequencing. It can easily be encoded in the syntax of Confined-λ.

On the other hand, the execution of the expression at B will cause an integer to be computed and sent to C over chBC. Note that we assume chBC to be a channel with mobility region $\{B,C\}$. The annotation of the created value shows us that the author of the code at B wishes to restrict the observability of this value to B and C. This wish, however, will not come true as the application of f to chBC at C, causes $\mathsf{secretBC}$ to be received at C and subsequently be sent to A.

(app-1) $$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[s : e_1\, e_2] \longrightarrow CI', P'[s : e_1'\, e_2]}$$

(app-2) $$\frac{CI, P[s : e_2] \longrightarrow CI', P'[s : e_2']}{CI, P[s : v\, e_2] \longrightarrow CI', P'[s : v\, e_2']}$$

(app-3) $$CI, P[s : (\mathsf{fn}^r\, x \Rightarrow e)\, v] \longrightarrow CI, P[s : e\{v/x\}]$$

(let-1) $$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[s : \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2] \longrightarrow CI', P'[s : \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_2]}$$

(let-2) $$CI, P[s : \mathsf{let}\, x = v\, \mathsf{in}\, e_2] \longrightarrow CI, P[s : e_2\{v/x\}]$$

(op-1) $$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[e_1\ op\ e_2] \longrightarrow CI', P'[s : e_1'\ op\ e_2]}$$

(op-2) $$\frac{CI, P[s : e_2] \longrightarrow CI', P'[s : e_2']}{CI, P[s : v\ op\ e_2] \longrightarrow CI', P'[s : v\ op\ e_2']}$$

(op-3) $$CI, P[s : v_1\ op\ v_2] \longrightarrow CI, P[s : v] \ \text{ where } v = v_1\ \mathbf{op}\ v_2$$

(chan) $$CI, P[s : \mathsf{chan}^r()] \longrightarrow CI \cup k^r, P[s : k^r] \ \text{ where } k^r \notin CI$$

(send-1) $$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[s : e_1\, !\, e_2] \longrightarrow CI', P'[s : e_1'\, !\, e_2]}$$

(send-2) $$\frac{CI, P[s : e_2] \longrightarrow CI', P'[s : e_2']}{CI, P[s : k^r\, !\, e_2] \longrightarrow CI', P'[s : k^r\, !\, e_2']}$$

(receive) $$\frac{CI, P[s : e_1] \longrightarrow CI', P'[s : e_1']}{CI, P[s : e_1 ?] \longrightarrow CI', P'[s : e_1' ?]}$$

(com) $$CI, P[s_1 : k^r\, !\, v][s_2 : k^r ?] \longrightarrow CI, P[s_1 : ()][s_2 : v]$$

Figure 5.2: Evaluation Rules

$$
\begin{array}{lll}
A: & \mathsf{let} & \mathsf{chAC} = \mathtt{chan}^{\{A,C\}}() \\
 & \mathsf{in} & \mathsf{let\ f} = \mathtt{fn}^{\{A,B,C\}}\mathsf{ch} \Rightarrow \quad \mathsf{let\ x} = (\mathsf{ch?}) \\
 & & \qquad\qquad\qquad\qquad\quad \mathsf{in\ chAC\,!\,x} \\
 & & \mathsf{in\ chTop\,!\,f;\ (chAC?)}
\end{array}
$$

$$
\begin{array}{ll}
B: & \mathsf{let\ secretBC} = \ldots^{\{B,C\}} \\
 & \mathsf{in\ chBC\,!\,secretBC}
\end{array}
$$

$$
\begin{array}{ll}
C: & \mathsf{let\ f'} = (\mathsf{chTop?}) \\
 & \mathsf{in\ f'\ chBC}
\end{array}
$$

Our approach in preventing violations such as this is to require the mobility region of a channel and that of the value it communicates to be identical. In our example, this would prevent secretBC from being sent over chAC.

Example 2: Region violation due to escaping values In this example, A is the origination site for function f which will be sent to B when the expression is executed. However, when applied at B, f will return a value which was originally intended to remain at A.

$$
\begin{array}{ll}
A: & \mathsf{let\ f} = \mathtt{fn}^{\{A,B\}}() \Rightarrow \ldots^{\{A\}} \\
 & \mathsf{in\ chAB\,!\,f}
\end{array}
$$

$$
B: \quad (\mathsf{chAB?})()
$$

Violations such as this can be prevented by requiring the mobility region of the result of a function to be at least as large as that of the function itself.

Example 3: Region violation due to remotely created values The function f below, which is allowed to be mobile between A and B, contains a channel allocation expression. The annotation shows us that this channel is intended to be used between A and C. However, when the function is applied at B, the channel will be created at B. According to our definition in Section 5.2.1, this will cause the values transmitted on this channel to be observable at B.

$$
\begin{array}{lll}
A: & \mathsf{let\ f} = \mathtt{fn}^{\{A,B\}}() \Rightarrow & \mathsf{let\ chAC} = \mathtt{chan}^{\{A,C\}}() \\
 & & \mathsf{in}\ \ldots \\
 & & \mathsf{in\ chAB\,!\,f} \\
 & & \\
B: & (\mathsf{chAB?})() &
\end{array}
$$

The problem here arises from the fact that functions are abstractions; the code enclosed by the function may be executed at a site which is different from the one where the function was created. Region violations due to this fact may be prevented by requiring the mobility region of the values created by the function to be a superset of all the sites where the function may be applied.

Example 4: Region violation due to transmission within a closure The function f below refers to the integer secretInt which is present in its definition environment. The annotation shows us that this integer is intended to remain at A. However, according to the dynamic semantics presented in Section 5.3.2, when the expression is executed secretInt will be substituted in the code of f and transmitted to B.

$$
\begin{array}{llll}
A: & \mathsf{let} & \mathsf{secretInt} = \ldots^{\{A\}} \\
 & \mathsf{in} & \mathsf{let\ f} = \mathtt{fn}^{\{A,B\}}() \Rightarrow \ldots \\
 & & (**\ \textit{do something with}\ \mathsf{secretInt}\ **) \\
 & & \mathsf{in\ chAB\,!\,f} \\
 & & \\
B: & (\mathsf{chAB?}) & ()
\end{array}
$$

To prevent region violations of this sort we need to address the dependency between the mobility region of a function and the values which occur in its definition environment.

5.4 Type system

The property called confinement in a mobility region was motivated in the preceding sections. The purpose of the type system presented in this section is to enforce that property. According to our system model, each site participating in the distributed computation provides an expression which is well-formed according to the syntax presented in Section 5.3. This expression is then analysed to check whether it is well-formed with respect to the rules of the type system given in Figure 5.4. We discuss these typing rules in detail in Section 5.4.2.

5.4.1 Semantic objects

The types of our system are pairs. The first component of the pair is a type in the conventional sense whereas the second component is used to record the mobility region. A type environment is a finite map from variables to types.

Mobility regions	r	$::=$	$\{s\} \mid r_1 \cup r_2$
Raw types	$\bar{\tau}$	$::=$	$\mathsf{unit} \mid \mathsf{int} \mid \mathsf{chan}[\bar{\tau}] \mid \tau_1 \to \tau_2$
Annotated types	τ	$::=$	$(\bar{\tau}, r)$
Type environments	Γ	$::=$	$[\,] \mid \Gamma[x \mapsto \tau]$

Definition 5.2 (L_s). L_s is a function used in typing rules to extract a mobility region from a type so that $L_s(\bar{\tau}, r) = r$.

We use two forms of judgements in our type system. A judgement of the form $\vdash \tau$ indicates that type τ is well-formed according to the rules given in Figure 5.3. We say that a type environment and a channel environment are well-formed if all of the types in their range are well-formed. A judgement of the form $r, \Gamma \vdash e : \tau$ indicates that the expression e can be assigned a well-formed type τ at any site within the mobility region r, by using the typing rules of Figure 5.4 and the well-formed type environment Γ. A distinctive point about our type system is the use of mobility regions in typing judgements along with typing and channel environments. This allows us to keep track of the static estimation of the origin and the execution site of an expression.

$$\vdash (\mathsf{unit}, r) \qquad \vdash (\mathsf{int}, r)$$

$$\frac{\vdash (\bar{\tau}, r)}{\vdash (\mathsf{chan}[\bar{\tau}], r)} \qquad \frac{\vdash \tau \quad \vdash \tau' \quad r \subseteq L_s(\tau')}{\vdash (\tau \rightarrow \tau', r)}$$

Figure 5.3: Well-formed Types

Definition 5.3 (Channel environment CE). A channel identifier is a finite map from channel identifiers to types. They are used to record the types of channels created in the course of the evaluation.

Definition 5.4 ($Type_{CE}$). $Type_{CE}$ is a family of functions indexed by channel environments. It is defined on constants of the language.

$$Type_{CE}(()^r) = (\mathsf{unit}, r) \quad Type_{CE}(n^r) = (\mathsf{int}, r) \quad Type_{CE}(k^r) = CE(k^r)$$

5.4.2 Typing rules

The typing rules are presented in Figure 5.4. Before looking at the rules in isolation, it would be helpful to note the following. The mobility region r used in the context of the judgement at the root of a typing derivation would be a singleton containing the name of the site at which the top-level expression is type-checked.

The role of the typing rule (con) for constants is to record in the type of a constant its specified mobility region. The side condition is used to ensure that no spurious declarations are made. For example s_1 cannot create a value and declare its mobility region to be s_2. The mobility region of a value is guaranteed to include its origination site.

The typing rule (id) is used for typing variables. The mobility region in the type of a variable is required to include the mobility region where the value of the variable may be found. This rule is crucial in preventing the kind of violation shown in Example 4; thanks to this rule we do not allow a value to appear in the closure of a function which may visit some sites where the value should not be observed. This rule also implies

(con)
$$\frac{Type_{CE}(c^r) = \tau \text{ where } \tau = (\bar{\tau}, r) \qquad r' \subseteq r}{r', \Gamma \vdash c^r : \tau}$$

(id)
$$\frac{\Gamma(x) = \tau \text{ where } \tau = (\bar{\tau}, r) \qquad r' \subseteq r}{r', \Gamma \vdash x : \tau}$$

(fn)
$$\frac{r, \Gamma[x \mapsto \tau] \vdash e : \tau' \qquad r' \subseteq r}{r', \Gamma \vdash \mathsf{fn}^r\, x \Rightarrow e : (\tau \rightarrow \tau', r)}$$

(app)
$$\frac{r', \Gamma \vdash e_1 : (\tau \rightarrow \tau', r) \qquad r', \Gamma \vdash e_2 : \tau}{r', \Gamma \vdash e_1\, e_2 : \tau'}$$

(let)
$$\frac{r', \Gamma \vdash e_1 : \tau \qquad r', \Gamma[x \mapsto \tau] \vdash e_2 : \tau'}{r', \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau'}$$

(op)
$$\frac{r', \Gamma \vdash e_1 : (\mathsf{int}, r_1) \qquad r', \Gamma \vdash e_2 : (\mathsf{int}, r_2)}{r', \Gamma \vdash e_1\, op\, e_2 : (\mathsf{int}, r_1 \cap r_2)}$$

(chan)
$$\frac{r' \subseteq r}{r', \Gamma \vdash \mathsf{chan}^r() : (\mathsf{chan}[\bar{\tau}], r)}$$

(send)
$$\frac{r', \Gamma \vdash e_1 : (\mathsf{chan}[\bar{\tau}], r) \qquad r', \Gamma \vdash e_2 : (\bar{\tau}, r)}{r', \Gamma \vdash e_1 ! e_2 : (\mathsf{unit}, r)}$$

(receive)
$$\frac{r', \Gamma \vdash e : (\mathsf{chan}[\bar{\tau}], r)}{r', \Gamma \vdash e? : (\bar{\tau}, r)}$$

Figure 5.4: Typing Rules

that a function cannot refer to a channel which was created outside its body unless this channel's mobility region is a superset of the mobility region of the function. This gives us a useful technical device for controlling the propagation of input/output capabilities caused by the mobility of functions.

There are three important aspects of the typing rule (fn) for functions. Firstly, functions are canonical expressions just as special constants. Their typing follows the same reasoning; we record the specified mobility region of a function in its type. The second aspect is to do with the fact that a mobility region of a function represents the sites which the function is allowed to visit. The well-formedness condition for function types ensures that a value returned by the function is allowed to be observed in the mobility region of the function itself. Region violations such as the one presented in Example 2 would be prevented by the presence of this condition. Finally, we need to take into consideration that the body of the function may be evaluated in a context different from that of the function abstraction. All we know about it is that it will be one of the sites that the function is allowed to visit. We type the body of an expression with respect to the mobility region of the function. Example 3 illustrates the significance of this final aspect.

The rules for applications and let expressions are quite straightforward. The typing rule (op) for primitive operations indicates that an operation op is allowed on integers only. The mobility region of an integer constructed from two integers using this operation is required to be the intersection of the regions of the components. Otherwise, a value could appear at a site which is outside its region through being a part of the composed value.

The concept of mobility region for a channel is not different from that of other data types. It represents the sites at which a channel is allowed to appear. The typing rule (chan) for channel allocation expressions ensures that a channel with a particular mobility region is allowed to be created at one of the sites in its mobility region.

The single most important point about the typing rules (send) and (receive) is that they force channels to carry values of the same type as themselves. We have motivated this rule by Example 1.

5.5 Formal properties

5.5.1 Confinement in a mobility region

We now state some properties of our system which will eventually lead us to formalize and prove the confinement in a mobility region property. Our type system guarantees that if an expression is well-typed at a particular site, it enjoys this property and therefore can be safely run at that site. We refer the interested readers to the Appendix for the details of the proofs.

Lemma 5.1 (Substitution). If $r, \Gamma[x \mapsto \tau] \vdash e : \tau'$ and $r, \Gamma \vdash v : \tau$ then $r, \Gamma \vdash e\{v/x\} : \tau'$.

Lemma 5.2 (Preservation of typability within a region). If $r, \Gamma \vdash v : (\bar{\tau}, r')$ then for any r'' such that $r'' \subseteq r'$ it is the case that $r'', \Gamma \vdash v : (\bar{\tau}, r')$.

Theorem 5.1 below shows that the type of an expression is preserved under transitions of the system.

Theorem 5.1 (Type Preservation). Assume that

- $CI, P[s : e] \longrightarrow CI', P'[s : e']$
- $CI = Dom(CE)$
- $r, [\,] \vdash e : \tau$.

Then there exists a CE' which extends CE such that $r, [\,] \vdash e' : \tau$ and $CI' = Dom(CE')$.

Proof. The proof is given by straightforward induction on the depth of inference of $CI, P[s : e] \longrightarrow CI', P'[s : e']$ by considering the possible forms of e. The cases referring to the third evaluation rule for applications and the second evaluation rule for let bindings appeal to Lemma 5.1. The case for communication makes use of the fact that a value has a unique type within its mobility region, regardless of the site at which the typing takes place. This is formalized by Lemma 5.2. $\square$

If expression e is well-typed as site s and it is evaluated at site s returning the value v as its result, then s is guaranteed to be within the mobility region specified for value v. Theorem 5.2 is the main result which we present in this chapter. We use the symbol $\rightarrow^*$ below to represent a sequence of transitions.

Lemma 5.3 (Conservative estimation). If $r, CE, \Gamma \vdash e : \tau$ then $r \subseteq L_s(\tau)$.

Lemma 5.4 (Consistency of labels). If $r, CE, \Gamma \vdash v : \tau$ then $L_d(v) = L_s(\tau)$.

Theorem 5.2 (Confinement in a mobility region). Assume for a closed e that

- $\{s\}, [\,] \vdash e : \tau$
- $CI, P[s : e] \rightarrow^* CI', P'[s : v]$.

Then $\{s\} \subseteq L_d(v)$.

Proof. The proof of the above theorem follows from Theorem 5.1 and Lemmas 5.3 and 5.4. Lemma 5.3 below shows that the type of an expression conservatively estimates the sites where the expression would be accepted as a result of the type-checking phase. If an expression e evaluates to a value v as is assumed in our theorem, by Theorem 5.1 and Lemma 5.3 we know that the label in the type of v will contain $\{s\}$. We also know by Lemma 5.4 that the mobility region annotations in the type of a value are consistent with the annotations provided as a part of the syntax, i.e. $L_d(v) = L_s(\tau)$. This allows us to conclude that $\{s\} \subseteq L_d(v)$. $\square$

5.5.2 Strong confinement

We want the property of confinement in a mobility region to hold for all the values which are observable in the sense defined in Section 5.2.1. For example, a value which is embodied in the code of a function is observable at a given site s if the function may appear at that site at any step during evaluation. The proof of Theorem 5.2 above is based on the consistency between the labels which annotate the values and their types. The definition of function L_d, however, does not look into the labels which may be enclosed in the body of a function.

We now define a function L_{rec} which examines an expression recursively to find the intersection of the mobility regions of its subexpressions. By using this we can show that the confinement property enforced by the type system is in fact stronger than the one formalized by Theorem 5.2.

Definition 5.5 (L_{rec}).

$L_{rec}(c^r) = r$	$L_d(\mathsf{fn}^r\, x \Rightarrow e) = r \cap L_{rec}(e)$
$L_{rec}(e_1 e_2) = L_{rec}(e_1) \cap L_{rec}(e_2)$	$L_{rec}(e_1\, op\, e_2) = L_{rec}(e_1) \cap L_{rec}(e_2)$
$L_{rec}(\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 = L_{rec}(e_1) \cap L_{rec}(e_2)$	$L_{rec}(\mathsf{chan}^r()) = r$
$L_{rec}(e_1 ! e_2) = L_{rec}(e_1) \cap L_{rec}(e_2)$	$L_{rec}(e_1 ?) = L_{rec}(e_1)$

The proposition below states that the expressions which may be enclosed in the code of a function are bound to have a mobility region which is at least as large as that of the function itself.

Proposition 5.1 (Strong confinement). If $r', \Gamma \vdash \mathsf{fn}^r\, x \Rightarrow e : (\tau \rightarrow \tau', r)$ then $r \subseteq L_{rec}(e)$.

Proof. It can be seen by inspecting the typing rules that r on the left hand side of the turnstile in a judgement either grows or remains the same as we go deeper in the typing derivation. By Lemma 5.3 we also know that each subexpression of e has a mobility region which is larger than or equal to r. By Definition 5.5 we can conclude that $r' \subseteq L_{rec}(e)$. □

We have defined the values of the language as expressions in canonical form. Hence, it follows from the proposition above that the values enclosed in a well-typed function have a mobility region at least as large as that of the function.

5.6 Related work

In this chapter, we have proposed a programming model where programmers can specify a policy about the flow of values and rely on the type system to enforce this policy. Adopting such a model can be motivated by several factors such as increasing locality, achieving better performance and controlling the flow of information in the system. In this section we will focus on the latter and view confinement in a mobility region as a secure information flow property.

Formulating and proving secure information flow properties for programs in the presence of code mobility is a challenging issue. There is a large body of work on exploiting type systems in this context. It is our view that a property such as confinement in a mobility region is a natural secure information property which arises in distributed computing with functions.

A wide range of languages have been subjected to study in the context of type-based approaches to security. These include imperative languages such as the ones considered in [SV97, SV98, SS00, ML99] and functional languages such as the Secure Lambda Calculus (SLam) [HR98a], an extension of the λ-calculus suitable for trust analysis [PØ97] and the Dependency Core Calculus [ABHR99]. Type systems for enforcing security properties in concurrent and mobile systems have also been studied in the framework of process calculi of the π-calculus family [HR00, CGG00], the ambient calculus [CGG99b] and the spi-calculus which is an extension of the π-calculus with cryptographic primitives [AG99]. We present here an in-depth discussion about the works which we consider as closely related to ours.

The SLam calculus The purely functional subset of the SLam calculus and Confined-λ are closely related. In both languages values are annotated to indicate their secrecy levels. In Confined-λ we call them mobility regions to make explicit that these annotations are used to restrict the mobility of values. However, there would be no essential difference if we interpreted them as sets of users who are allowed to observe these values as suggested by the SLam calculus.

The annotations of the SLam calculus, called security properties, are more expressive than the annotations of Confined-λ. This is mainly due to the differences in our motivations. The SLam calculus aims at capturing refined notions of security by exploiting rich type structures. Taking a complementary approach, the aim of this work is to establish a simple notion of secrecy for distributed computation by remaining close in spirit to conventional type systems for functional languages.

The work on the SLam calculus distinguishes between direct readers and indirect readers of values in order to be able to address indirect information flows such as those caused by branching on high security vales in conditional expressions. The annotations of the SLam calculus values are tuples, where the first and the second components specify respectively its authorised direct readers and indirect readers. The authors also introduce dual notions to readers and indirect readers; creators and indirect creators represent users who might have created the objects directly or indirectly. Mobility region annotations of Confined-λ can be likened to reader annotations of the SLam calculus with no distinction between direct readers and indirect readers.

A formal study of the correspondence between the functional subsets of the SLam calculus restricted to reader annotations and Confined-λ with mobility region annotations could be a future work in its own right. We now discuss the typing of a function abstraction and application in both systems to illustrate the similarities and differences of the adopted approaches.

The following is the typing rule for functions in the SLam calculus. The annotation κ is a tuple of the form (r, ir) where r is the direct reader annotation and ir is the indirect reader annotation. A global condition on the type system requires r to indicate a higher secrecy level than ir.

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.e : \tau_2)_\kappa : (\tau_1 \rightarrow \tau_2, \kappa)}$$

A function which is typed with respect to the rule above can only be applied by those readers whose secrecy level is at least as high as the one indicated by the reader annotation r of the function. In order to enforce this, application expressions are annotated with the security levels of programmers and a side condition is used in the typing rule to check that the programmer is authorised to apply the function.

$$\frac{\Gamma \vdash e_1 : (\tau_1 \rightarrow \tau_2, (r, ir)) \quad \Gamma \vdash e_2 : \tau_1 \quad r \sqsubseteq r'}{\Gamma \vdash (e_1\, e_2)_{r'} : \tau_2 \bullet ir}$$

It is the responsibility of the trusted compiler to ensure that programmers provide annotations which are consistent with their actual security levels. The rule above also shows that the result of the application is to be protected at the security level $\tau_2 \bullet \kappa$. However, this is not relevant to our present discussion.

We now look at the typing rules for functions and applications in Confined-λ. Note that we have omitted the channel environment CE in the typing judgements as we are comparing the sequential subsets only. In order to facilitate the comparison of the two systems we could read a judgement of the form $r, \Gamma \vdash e : \tau$ as follows: an expression e provided by an author who is a member of r can be assigned the type τ.

$$\frac{r, \Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \quad r' \subseteq r}{r', \Gamma \vdash \mathsf{fn}^r\, x \Rightarrow e : (\tau_1 {\rightarrow} \tau_2, r)}$$

Informally speaking, the role played by the typing rule for applications in the SLam calculus is assigned to the typing rule for functions in our type system. The only essential difference is that we force the annotation of the function to include its author. This reflects our view that it would be natural to allow the use of a function by its own author. The body of the function above can be executed by the users in r, hence its typing takes place in a context which includes r. This is how we enforce authorised access to the code of the function. Our approach eliminates the need for annotating application expressions as in the SLam calculus. The typing rule for applications in Confined-λ is as follows:

$$\frac{r', \Gamma \vdash e_1 : (\tau {\rightarrow} \tau_1, r) \quad r', \Gamma \vdash e_2 : \tau}{r', \Gamma \vdash e_1\, e_2 : \tau_1}$$

The SLam calculus and Confined-λ adopt different communication models. The SLam calculus assumes a shared memory where the communication is based on mutable data structures. Confined-λ, however, supports communication by message passing as this is a more natural method of communication in distributed systems. We also note that communication based on shared memory is inherently asynchronous whereas the sender of a message and its receiver must synchronise in Confined-λ. These differences preclude a direct comparison of the two type systems as we were able to do for the functional core. It would be interesting to formalise an extension of the SLam calculus with distributed shared memory and devise a type system for that language which enjoys the secrecy property of this paper. We would regard any similarity between this type system and that of Confined-λ as an encouraging sign that secrecy can be dealt with using similar techniques in functional languages which adopt different communication models.

Process calculi The channel-based communication model and the presence of explicit localities relate Confined-λ to process calculi for concurrent and distributed computation, particularly to higher-order extensions of the π-calculus such as the ones presented in [YH99, Tho89]. We are not aware of any other work on these calculi which focuses on a notion of secure information flow such as ours. On the other hand, enforcement of secure information flow by typing has been investigated in the framework of process calculi whose relation to Confined-λ is relatively indirect. For example, a type system which is aimed at detecting information leaks to the environment has been developed for the spi-calculus [AG99]. This work provides insights into the questions of formulating and proving secrecy properties for concurrent systems where interaction capabilities evolve dynamically. However, the languages Confined-λ and the spi-calculus differ in several respects. The spi-calculus is a calculus which is particularly suited for the description and analysis of cryptographic protocols. It extends the core of the π-calculus with cryptographic primitives. There is no direct way of expressing the communication of functions between processes. The characteristic feature of Confined-λ, however, is that it offers a simple model for distributed computation which is based on the mobility of functions between concurrently executing expressions at remote sites.

A language which is unlike all of the other languages cited above but which can be regarded as related to Confined-λ is uPLAN [KGA00]. This language has been defined to provide a formal model of computation in active networks. It assumes a fixed set of sites and the computation is based on the mobility of functions. In these respects it is similar to Confined-λ. However, the mobility in uPLAN is based on a facility for remote evaluation of functions at explicitly specified locations. The authors propose a notion of secrecy inspired by the spi-calculus. The proof of their secrecy property does not depend on a type-based approach such as ours.

5.7 Concluding remarks

The issues related to information flow due to covert storage channels or covered timing channels are not within the scope of this chapter. It would be interesting to investigate these issues for an extension of Confined-λ with conditional expressions. The analogy between secrecy levels and mobility regions should allow us to benefit from the works which focus primarily on noninterference properties such as the ones studied in [HR98a, VS98, PC00] and which focus on timing leaks [Aga00].

In the process calculi framework, some researchers have exploited type systems to enforce locality conditions on the use of capabilities [YH99, Sew98]. It is possible to regard confinement in a mobility region as a locality condition; we restrict the distribution of communication of capabilities to a part of the system as a means of controlling the flow of information within the system. Distinguishing between input and output capabilities, as in [YH99, Sew98], could allow us to explore more refined notions of secure information flow. This remains as an interesting direction for future work.

Chapter 6

Noninterference and Mobile Functions

In this chapter we continue our exploration of secure information flow in multi-user distributed systems. Of particular interest to us are systems in which data and users are classified such that the security class of a datum reflects its confidentiality level and the security class of a user determines which data he is authorized to observe.

A natural security requirement for systems of this kind is to prevent users from accessing data which they are not authorized to observe. Access control, however, addresses only a single aspect of secure information flow. It should not be overlooked that some users may exploit indirect means to obtain confidential information rather than attempting to access it directly. One can think of a scenario in which a user writes a program whose behaviour depends on the values of particular variables in the environment. Even if the user himself is not allowed to access these variables directly, he can get another user with the required permissions to execute the program. By observing its behaviour he can then infer the values of the variables. Note that such leakage of information can be caused by a malicious cooperation among users or inadvertently.

A programming language whose legal programs are guaranteed not to cause a certain class of information flows would contribute to preventing security violations of this kind. In this chapter, we investigate the design of a language based on Mobile-λ which is targeted at computation in systems with classified data and users. We first determine a confidentiality property based on *noninterference*. We then propose a variant of the type and effect system presented in Chapter 3 to statically enforce this property.

6.1 Noninterference

Formulating confidentiality properties which account for the absence of undesirable information flows is a challenging task, let alone developing methods to enforce them. There is a vast literature on *security models* for describing confidentiality requirements of systems. A comprehensive survey can be found in [McL94]. Noninterference emerges as a useful concept in interface models for confidentiality which specify restrictions on the input/output relation of systems.

6.1.1 A general characterization

The first appearance of the concept of noninterference in the security literature is attributed to Goguen and Meseguer [GM82].

> One group of users, using a certain set of commands, is noninterfering with another group of users if what the first group does with those commands has no effect on what the second group of users can see.

In their seminal work the authors presented an approach to designing secure systems which is based on modelling a system by an automaton and defining security policies as sets of noninterference assertions which can then be verified by appropriate proof techniques. Gougen and Meseguer's work has had a significant impact on the subsequent characterizations of secure information flow. This is not due to the particular formalism the authors used but to the fact that noninterference is a simple and intuitive notion. Proving that a system is noninterfering is not dependent on the availability of an automaton model of a system. Different formulations of noninterference and proof techniques exist for different formalisms [Den76, FG95, RS01].

For deterministic systems, noninterference is considered as a satisfactory notion of secure information flow on which enforcable security policies can be based. Its generalization to nondeterministic systems, however, is not straightforward. Illuminating discussions on the topic and examples can be found in [McL94, VS98].

6.1.2 A restriction on the input/output relation

In this chapter we assume that there are only two security classes in a system: H (high) and L (low). Secret values belong to the class H and public values belong to the class L. Users who belong to the security class H can observe any value whereas users who belong to the security class L can observe public values only. In this setting, the flow of information from the class H to the class L is considered as undesirable.

A noninterference property typically states that high-level inputs to the system cannot interfere with low-level outputs. In other words, the values of public outputs should not depend on the values of secret inputs. The characteristics of the system in question and what is assumed to be observable by whom influences the formulation of a noninterference property.

6.1.3 Closer look at Mobile-λ

Noninterference is concerned with high-level inputs and low-level outputs of a system. In order to talk about noninterference for computation with languages of the Mobile-λ family it is necessary to make precise the sense in which the terms input and output are used.

If the computation in a system consisted of a single thread of control with no reference to the environment – this corresponds to the application of a closed function in our case – what is meant by input and output would be straightforward. We would be using the term input for function arguments and the term output for function results. However, we have multiple concurrently executing functions which can communicate with each other and access the resources in their environment. The inputs to the system are not merely the arguments to the top-level functions but also other values which may flow into the function. The values which are received on channels and the values which are bound to the free identifers of the function should also be taken into consideration. Similarly, the outputs of the system include values which are sent on communication channels as well as those returned by functions.

A noninterference property for mobile functions should at the very least express that the public results returned by functions and the public values which flow out of functions on communication channels do not depend on the secret values which flow into functions. The noninteference property that we enforce by means of a type system in Section 6.3 formalizes this idea.

It is possible to propose noninterference properties of relative strength. For example, if we assume that L users can infer H information to a high degree of certainty by observing the timing behaviour or nontermination of programs, or blocked communications, the formulation of the noninterference property would be different to the setting where these assumptions do not hold. In this chapter, we concern ourselves only with computations which terminate by producing a result. We also assume that users cannot infer confidential information by observing the time taken for computation.

6.1.4 Conditional expressions

A challenging point in the enforcement of a noninterference property arises from the presence of conditional expressions in programs. There is always an implicit flow from the guard of a conditional expression to its branches. Let us suppose that `secretBool` is an identifier which refers to a boolean value of the security class H in the following conditional expression.

```
if secretBool then true else false
```

If a user of security class H were to evaluate this expression and a user of the security class L who knows the code were allowed to observe the result, the value of the `secretBool` would be leaked.

6.1.5 Example

In the following sections we will define a meta-language which will provide a formal framework for stating and proving a noninterference property for mobile functions. Prior to that we present an example to give a more clear idea of undesirable information flows in the computational model we are considering. We focus on a simple client-server application coded in a language of the Mobile-λ family.

```
Client:  let  secretInt = 500
         in   (sq?)  secretInt

Server:  let  gt100 = chan()
         in   let square = fn x   => if x > 100
                                     then ( gt100 ! true; x*x )
                                     else ( gt100 ! false; x*x )
              in ( sq ! square ;  gt100?  )
```

The intention of the client is to take the square of an integer which is confidential to the client. The code necessary for this operation is made available by the server. All that is needed by the client is to make a request for the code of the function `square` on the channel `sq` and apply the received code to the integer.

The function `square` provided by the server has, however, been coded in a rather malicious way. It not only performs the requested operation but also leaks information about the argument it is applied to at the client's side. The closure of the function `square` contains the channel `gt100` allocated at the server's side which is used to maintain a connection back to the server. After sending `square` to the client, the server listens on this channel. The code of the function `square` includes a test on the argument which returns true if the argument is greater than 100, false otherwise. The users who know the code of the server and can observe what is returned on the channel `gt100` can infer whether the client applied the function `square` to an integer greater than 100.

The language introduced in Section 6.2 has two major aims. Firstly, it facilitates the static declaration of security classes of the values which will be generated during evaluation. Secondly, a typing discipline is imposed to guarantee that certain kinds of information flows cannot occur during evaluation. For example, if the value of `secretInt` is to be known only to high-level users, this would be declared by the client by annotating it with the appropriate label. If the server wants the value received on the channel `gt100` to be observable by low-level users, the expression which allocates the channel `gt100` should be annotated accordingly. The type system of Section 6.3 is based on the idea of using these annotations to detect undesirable flows of information from the security class H to the class L.

6.2 Secure Mobile-λ

In this section we introduce the language Secure Mobile-λ which is derived from Mobile-λ. In contrast to the uniform nature of values in Mobile-λ with respect to confidentiality, computation with Secure Mobile-λ involves values which may belong to different security classes.

6.2.1 Abstract syntax

The main characteristic of the abstract syntax of Secure Mobile-λ is that all expressions which are in canonical form and channel allocation expressions are labelled with security classes. This is intended to assist with tracing the security classes of values throughout the computation.

Labels	l	::=	$H \mid L$	
Expressions	e	::=	c^l	annotated constants
			$\mid \mathsf{fn}^l\, x \Rightarrow e$	annotated functions
			$\ldots$	expressions of the Core
			$\mid \mathsf{chan}^l()$	channel allocation
			$\mid e_1!e_2$	send
			$\mid e?$	receive

Figure 6.1: Abstract Syntax for Secure Mobile-λ

Definition 6.1 (Ordering on secrecy labels). Secrecy labels can be partially ordered with respect to the relation $\sqsubseteq$ such that $L \sqsubseteq H$. The set of secrecy labels $SL = \{H, L\}$ and the ordering $\sqsubseteq$ form a two-point lattice with L and H as the least and the greatest elements respectively.

6.2.2 Dynamic semantics

We consider a simple system which consists of two remote sites each of which has a trusted Secure Mobile-λ compiler. The most important assumption about the system is that it is deterministic. Each site in the system provides a standard set of values (ubiquitous values) as a part of its computational environment. Ubiquitous values include services coded as functions. Throughout our discussion, we assume that ubiquitous functions do not introduce nondeterminism to the system or have side-effects which violate the confidentiality requirements which we impose on the user-defined functions.

Values of Secure Mobile-λ consist of primitive values, channel identifiers and function closures, represented by the semantic objects c^l, k and $\langle l, E, x, e\rangle$ respectively. The dynamic representation of an object encodes the secrecy label of the expression which leads to its construction. A channel identifier k is represented by a tuple (l, s, i) where l is a security label, s is the site where the channel is allocated and i is a freshly generated number at each invocation of a channel allocation expression. We write $Label(v)$ for the secrecy label of a value.

Evaluation environment	E	::=	$[\,] \mid E[x \mapsto v]$	
Values	v	::=	c^l	constants
			$\mid k$	channel identifers
			$\mid \langle l, E, x, e\rangle$	function closures

We also use semantic objects which describe the communication behaviour of an expression; a stands for a communication action drawn from the set $\{\mathsf{new}\, k, k\,!\,v, k\,?\,v\}$ representing the allocation of a channel and sending and receiving values over a channel respectively. The name w stands for a sequence of communication actions. Note that a purely sequential computation which does not involve any communication is represented by ε.

Actions	a	::=	$\mathsf{new}\, k$	channel allocation
			$\mid k\,!\,v$	send
			$\mid k\,?\,v$	receive
Action sequences	w	::=	ε	the empty sequence
			$\mid a.w$	sequence of actions

6.2.2.1 Top-level rule and matching

Secure Mobile-λ does not include a construct for parallel composition of expressions. The concurrent activity is revealed in the composition of evaluations at each site. Figure 6.2 shows the evaluation of a system in which expression e_1 is evaluated at site s_1 and expression e_2 is evaluated at site s_2. We represent the top-level environment as a pair of environments and the result produced by the system as a pair of values. The top-level evaluation decomposes into individual evaluations at each site. A judgement of the form $E \vdash e \overset{w}{\Longrightarrow} v$ states that the expression e evaluates to value v against an environment E incurring the sequence of actions represented by w. The dynamic semantics rules for the evaluation of an expression are further explained in Section 6.2.2.2.

$$\frac{E_1 \vdash_{s_1} e_1 \overset{w_1}{\Longrightarrow} v_1 \qquad E_2 \vdash_{s_2} e_2 \overset{w_2}{\Longrightarrow} v_2 \qquad w_1 \mid\mid w_2 \hookrightarrow w}{(E_1, E_2), (s_1[e_1] \mid\mid s_2[e_2]) \overset{w}{\Longrightarrow} (v_1, v_2)}$$

Figure 6.2: Concurrent Evaluation

The communication in Secure Mobile-λ is synchronous. The synchronization between the expressions is specified by a collection of matching rules between communication action sequences given in Figure 6.3. Informally speaking, at the end of the evaluation, we can observe that new channels have been allocated. Particular values transmitted on these channels will have been used in producing the resulting value, however, we treat their transmissions as internal actions of the system and abstract away from them. The predicate $IOmatch$ below is defined on pairs of action sequences. We will refer to this predicate later to express that the system consisting of two sites is closed. That is, all the communication actions are internal to the system.

Definition 6.2 (IOmatch). $IOmatch(w_1, w_2)$ if and only if $w_1 \mid\mid w_2 \hookrightarrow w$ and all annotations in w are of the form new k for some k.

(1)	$\varepsilon \mid\mid \varepsilon \hookrightarrow \varepsilon$
(2)	$\dfrac{w_1 \mid\mid w_2 \hookrightarrow w}{w_1.a \mid\mid w_2 \hookrightarrow w.a}$
(3)	$\dfrac{w_1 \mid\mid w_2 \hookrightarrow w}{w_1 \mid\mid w_2.a \hookrightarrow w.a}$
(4)	$\dfrac{w_1 \mid\mid w_2 \hookrightarrow w}{w_1.k!val \mid\mid w_2.k?val \hookrightarrow w}$
(5)	$\dfrac{w_1 \mid\mid w_2 \hookrightarrow w}{w_1.k?val \mid\mid w_2.k!val \hookrightarrow w}$

Figure 6.3: Action Matching

6.2.2.2 Evaluation rules

The intuition behind the evaluation rules are similar to those of Mobile-λ. However, our style of defining the semantics of Secure Mobile-λ is different. We do not specify the intermediate steps of the evaluation but rather define a relation between expressions and their values in the style of [Ler92].

Functions The rule (fn) for the evaluation of function abstractions is similar to the evaluation rule with the same name for Mobile-λ. Before being enclosed in the function closure the current environment is narrowed down by removing the ubiquitous values from the environment. For an application expression to evaluate successfully, the first expression must be a function closure. The body of the function enclosed in the closure is evaluated against an environment which is obtained by extending the environment part of the closure with the binding of the argument and the ubiquitous environment. The reader is referred to Chapter 3 for formal definitions of the operations such as $\downharpoonright$ and $\oplus$ on semantics objects.

Primitive operations Binary operators which are ranged over by *op* can be either arithmetic or relational. The application of a binary operator on two values results in a value whose secrecy level is the highest of those of its operands. That is to say if

(con) $E \vdash c^l \stackrel{\varepsilon}{\Longrightarrow} c^l$

(id) $E \vdash x \stackrel{\varepsilon}{\Longrightarrow} E(x)$

(fn) $E \vdash \mathsf{fn}^l\, x \Rightarrow e \stackrel{\varepsilon}{\Longrightarrow} \langle l, E, x, e\rangle$ where $E' = E \downharpoonright FI(\mathsf{fn}^l\, x \Rightarrow e)$

(app)
$$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} \langle l', E, x, e\rangle \quad E \vdash e_2 \stackrel{w_2}{\Longrightarrow} v \qquad E_u \oplus E'[x \mapsto v] \vdash e \stackrel{w_3}{\Longrightarrow} v'}{E \vdash e_1\, e_2 \stackrel{w_1.w_2.w_3}{\Longrightarrow} v'}$$

(op)
$$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} v_1 \quad E \vdash e_2 \stackrel{w_2}{\Longrightarrow} v_2 \quad v = v_1\, \mathbf{op}\, v_2}{E \vdash e_1\, op\, e_2 \stackrel{w_1.w_2}{\Longrightarrow} v}$$

(if-t)
$$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} \mathsf{true}^l \quad E \vdash e_2 \stackrel{w_2}{\Longrightarrow} v}{E \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \stackrel{w_1.w_2}{\Longrightarrow} v}$$

(if-f)
$$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} \mathsf{false}^l \quad E \vdash e_3 \stackrel{w_2}{\Longrightarrow} v}{E \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \stackrel{w_1.w_2}{\Longrightarrow} v}$$

(let)
$$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} v_1 \quad E[x \mapsto v_1] \vdash e_2 \stackrel{w_2}{\Longrightarrow} v_2}{E \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \stackrel{w_1.w_2}{\Longrightarrow} v_2}$$

(chan) $E \vdash \mathsf{chan}^l() \stackrel{\mathsf{new}\, k}{\Longrightarrow} k$ where k new

(send)
$$\frac{E \vdash e_1 \stackrel{w_1}{\Longrightarrow} k \quad E \vdash e_2 \stackrel{w_2}{\Longrightarrow} v \quad k = (l, s, i)}{E \vdash e_1 ! e_2 \stackrel{w_1.w_2.k!v}{\Longrightarrow} ()^l}$$

(receive)
$$\frac{E \vdash e \stackrel{w}{\Longrightarrow} k}{E \vdash e? \stackrel{w.k?v}{\Longrightarrow} v}$$

Figure 6.4: Sequential Evaluation Rules

$v = (v_1 \textbf{ op } v_2)$ for some **op** where $Label(v_1) = l_1$ and $Label(v_2) = l_2$ then $Label(v) = l_1 \sqcup l_2$. For example, an integer which is obtained by adding a public and a secret integer would be treated as a secret integer.

Expressions involving communication The evaluation rule (chan) states that a channel identifier which is allocated dynamically must be globally unique. We have already discussed that a channel identifier is represented by a tuple (l, s, i). Since i is freshly generated each time a new channel is allocated at site s, s and i together can guarantee the uniqueness of an identifier. The evaluation rules (send) and (receive) are straightforward.

6.3 Type system

The aim of the type system presented in this section is to guarantee that the values of security class L which flow out of the system during the evaluation of a well-typed Secure Mobile-λ do not depend on the values of security class H which flow into the system.

6.3.1 Semantic objects

The type system builds on similar ideas to the type system presented for Mobile-λ in Chapter 3. The structure of the types and effects are almost identical. The crucial difference lies in the purpose served by the labels. In that type system the labels stood for the identities of values, in this one they stand for their security classes.

Another difference is with regard to polymorphism. The polymorphism in the type system of this section is more restricted; we allow function types to be parametric in secrecy labels only. This is motivated by our wish to keep the type system as simple as possible while showing how ML-style polymorphism can be adapted to increase its flexibility.

Secrecy labels	l	::=	$H \mid L \mid \gamma \mid l_1 \sqcup l_2$
Raw types	$\bar{\tau}$	::=	$\mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{chan}[\bar{\tau}] \mid \tau_1 \xrightarrow{\kappa} \tau_2$
Types	τ	::=	$(\bar{\tau}, l)$
Effect	κ	::=	$\emptyset \mid \{\mathtt{new}\, l\ \mathsf{for}\ \bar{\tau}\}$
			$\mid \{\mathsf{send}\ \bar{\tau}\ \mathsf{on}\ l\} \mid \{\mathsf{recv}\ \bar{\tau}\ \mathsf{on}\ l\} \mid \kappa_1 \cup \kappa_2$
Type schemes	σ	::=	$\forall \vec{\gamma}.\tau$
Type environments	Γ	::=	$[\,] \mid \Gamma[x \mapsto \sigma]$

Definition 6.3 (Well-formed function types). We write $Label(\tau)$ for the label of a type; if $\tau = (\bar{\tau}, l)$ then $Label(\tau) = l$. A function type $(\tau \xrightarrow{\kappa} \tau', l)$ is well-formed if $l \sqsubseteq Label(\tau')$.

Definition 6.4 (TypeOf). The types of the basic constants unit, integers and booleans do not depend on the typing context and are defined as follows:

$$TypeOf(()^{l}) = (\mathsf{unit}, l) \qquad TypeOf(n^{l}) = (\mathsf{int}, l)$$
$$TypeOf(\mathsf{true}^{l}) = (\mathsf{bool}, l) \quad TypeOf(\mathsf{false}^{l}) = (\mathsf{bool}, l)$$

6.3.2 Typing rules

The typing rules are given in Figure 6.5. An important design decision which has an impact on the whole of the type system regards communication. It can be seen in the typing rules (send) and (receive) that we allow channels to carry values of the same secrecy class as themselves.

Keeping this in mind we can now go on to explain the more interesting rules. Those are the rules for function abstractions and conditional expressions. We choose to start with the (if) rule since this will make it easier to understand why we need a side condition in the typing rule (fn) for functions.

Let us consider a simple conditional expression such as the one which appeared in Section 6.1. The secretBool below is an identifier of a secret boolean and the branches return public values.

$$\mathsf{if\ secretBool\ then\ true}^{L}\mathsf{else\ false}^{L}$$

(con) $\Gamma \vdash c^l : \mathit{TypeOf}(c^l), \emptyset$

(id) $$\frac{\Gamma(x) = \sigma \quad \sigma \succ \tau}{\Gamma \vdash x : \tau, \emptyset}$$

(fn) $$\frac{\Gamma[x \mapsto \tau] \vdash e : \tau' \quad \mathit{Safe}(\kappa, l)}{\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e : (\tau \xrightarrow{\kappa} \tau', l), \emptyset}$$

(app) $$\frac{\Gamma \vdash e_1 : (\tau \xrightarrow{\kappa} \tau', l'), \kappa' \quad \Gamma \vdash e_2 : \tau, \kappa''}{\Gamma \vdash e_1\, e_2 : \tau', \kappa \cup \kappa' \cup \kappa''}$$

(op) $$\frac{\Gamma \vdash e_1 : (\bar{\tau}, l), \kappa \quad \Gamma \vdash e_2 : (\bar{\tau}, l'), \kappa' \quad op : (\bar{\tau} * \bar{\tau}) \rightarrow \bar{\tau}'}{\Gamma \vdash e_1\, op\, e_2 : (\bar{\tau}', l \sqcup l'), \kappa \cup \kappa'}$$

(if) $$\frac{\begin{array}{l}\Gamma \vdash e_1 : (\mathsf{bool}, l), \kappa \\ \Gamma \vdash e_2 : (\bar{\tau}, l'), \kappa' \quad \Gamma \vdash e_3 : (\bar{\tau}, l'), \kappa'' \\ l \sqsubseteq l' \quad \mathit{Safe}(\kappa', l) \quad \mathit{Safe}(\kappa'', l)\end{array}}{\Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : (\bar{\tau}, l'), \kappa \cup \kappa' \cup \kappa''}$$

(let) $$\frac{\Gamma \vdash e_1 : \tau, \kappa \quad \Gamma[x \mapsto \mathit{Gen}(\Gamma, \kappa, \tau)] \vdash e_2 : \tau', \kappa'}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau', \kappa \cup \kappa'}$$

(chan) $\Gamma \vdash \mathsf{chan}^l() : (\mathsf{chan}[\bar{\tau}], l), \{\mathtt{new}\, l\ \mathsf{for}\ \bar{\tau}\}$

(send) $$\frac{\Gamma \vdash e_1 : (\mathsf{chan}[\bar{\tau}], l), \kappa \quad \Gamma \vdash e_2 : (\bar{\tau}, l), \kappa'}{\Gamma \vdash e_1 ! e_2 : (\mathsf{unit}, l), \kappa \cup \kappa' \cup \{\mathsf{send}\, \bar{\tau}\, \mathsf{on}\, l\}}$$

(receive) $$\frac{\Gamma \vdash e : (\mathsf{chan}[\bar{\tau}], l), \kappa}{\Gamma \vdash e? : (\bar{\tau}, l), \kappa \cup \{\mathsf{recv}\, \bar{\tau}\, \mathsf{on}\, l\}}$$

(subs) $$\frac{\Gamma \vdash e : \tau, \kappa \quad \kappa \subseteq \kappa'}{\Gamma \vdash e : \tau, \kappa'}$$

Figure 6.5: Typing Rules

Expressions such as this one should obviously be ruled out because the value of this expression which is observable by low-level users depends on a high-level value. The side condition $l \sqsubseteq l'$ is present in the rule (if) for this reason. This side condition is, however, not sufficient to detect all kinds of information flows that we wish to eliminate. The communication capabilities of the branches need also to be taken into account. Let us suppose that ch is the identifier of a public channel in the example presented below.

$$\begin{array}{lll} \mathsf{if\ secretBool} & \mathsf{then} & \mathsf{let\ y} = \mathsf{ch\ !\ true}^{L} \\ & & \mathsf{in\ true}^{H} \\ & \mathsf{else} & \mathsf{let\ y} = \mathsf{ch\ !\ false}^{L} \\ & & \mathsf{in\ false}^{H} \end{array}$$

Even though the result of this expression is a high-level value which is observable only by high-level users, the information would be leaked due to the communication on public channels. A low-level user who listens on channel ch would be able to infer the value of secretBool. The predicate *Safe* in the typing rule (if) is used to detect the presence of communication capabilities such as this. Branches of an expression can only send on channels which are at least as secret as the guard of that expression.

Definition 6.5 (Safe flow). $\mathit{Safe}(\kappa, l)$ if for all $\{\mathsf{send}\,\bar{\tau}\,\mathsf{on}\,l'\} \subseteq \kappa$ it is the case that $l \sqsubseteq l'$.

Functions are allowed to send on channels which are at least as secret as themselves; this is also imposed by the presence of the predicate *Safe* in the typing rule (fn). The following example justifies the need for this side condition.

$$\begin{array}{ll} \mathsf{if\ secretBool} & \mathsf{then}\ (\mathsf{fn}^{H}\,\mathsf{x} \Rightarrow \mathsf{let\ y} = \mathsf{ch\ !\ true}^{L}\ \mathsf{in\ true}^{H}) \\ & \mathsf{else}\ (\mathsf{fn}^{H}\,\mathsf{x} \Rightarrow \mathsf{let\ y} = \mathsf{ch\ !\ false}^{L}\ \mathsf{in\ false}^{H}) \end{array}$$

The potentially harmful communication capabilities on public channels are now hidden within a function abstraction. If the value of secretBool is, for example, true the expression will return a function which sends true on the public channel ch. By applying this function to an appropriate argument a similar effect to the preceding example can be incurred. What is leaked in a single step in the preceding example is leaked in two steps in this one.

It is important to remember also that the secrecy class of the result of a function is required to be at least as secret as that of the function itself. It is through the well-formedness condition in Definition 6.3 on function types that we enforce this. Without this condition a public value could be obtained by the application of a secret function present in the environment. This would conflict with our requirement that public values should not depend on secret values.

The type generalisation takes place in the typing rule (let). We omit the definition of the operation *Gen* as it is a straightforward adaptation of the operation with the same name which appears in Chapter 3.

6.3.2.1 Example

If we did not allow functions to be parametric in security labels, the type system would be rather too restrictive. The following example illustrates the use of polymorphism. Suppose that $\mathsf{secretInt}$ and $\mathsf{publicInt}$ are identifiers of a secret integer and a public integer respectively and that we would like to type-check the following expression in the environment $\Gamma = [\mathsf{secretInt} \mapsto (\mathsf{int}, H), \mathsf{publicInt} \mapsto (\mathsf{int}, L)]$.

$$\begin{array}{l} \mathsf{let\ square} = \mathsf{fn}^L \mathsf{x} \Rightarrow \mathsf{x} * \mathsf{x} \\ \mathsf{in\ (square\ publicInt)} * (\mathsf{square\ secretInt}) \end{array}$$

If we did not allow types to be parametric in security labels, we would not be able to derive a type for this expression. It would be necessary to write two versions of the function square; one to be applied to secret integers, one to be applied to public ones. We can, however, derive the type $\forall\gamma.((\mathsf{int},\gamma)\xrightarrow{\emptyset}(\mathsf{int},\gamma),L)$ for the function square. The type of the entire expression is then (int, H) where the type of the subexpression $(\mathsf{square\ publicInt})$ is (int, L) and that of $(\mathsf{square\ secretInt})$ is (int, H).

6.4 Formal properties

In this section we first prove the consistency of the dynamic and the static semantics. We then state a noninterference property for mobile functions in a deterministic computational model and prove that it is enjoyed by the well-typed Secure Mobile-λ programs.

6.4.1 Consistency

In defining the dynamic semantics of Secure Mobile-λ we adopted an approach which is different from those of the preceding chapters. The proof method we use, which follows the approach of [Ler92] is hence unlike those of the preceding chapters.

Definition 6.6 (Channel environment). A channel environment is a finite map from channel identifiers to types $CE ::= [] \mid CE[k \mapsto \tau]$.

Definition 6.7 (Extension). Let CE and CE' be two channel environments. CE' extends CE, written as $CE \sqsubseteq CE'$ if $Dom(CE) \subseteq Dom(CE')$ and $CE(k) = CE'(k)$ for all $k \in Dom(CE)$.

We now define a relation between the values of the dynamic semantics and types. We write $CE \models v : \tau$ to mean that a value v has type τ where the types of the dynamically allocated channels are recorded in the channel environment CE.

Definition 6.8 ($CE \models v : (\bar{\tau}, l)$).

$CE \models c^l : TypeOf(c^l)$

$CE \models k : (\mathsf{chan}[\bar{\tau}], l)$ if $k \in Dom(CE)$ and $CE(k) = (\bar{\tau}, l)$

$CE \models \langle l, E, x, e \rangle : (\tau \xrightarrow{\kappa} \tau', l)$ if there exists a Γ such that
$\Gamma \vdash \mathtt{fn}^l\, x \Rightarrow e : (\tau \xrightarrow{\kappa} \tau', l), \emptyset$ and $CE \models E : \Gamma$

$CE \models v : \forall \vec{\gamma}.\tau$ if $CE \models v : \theta\tau$
for any substitution θ defined on $\vec{\gamma}$.

This relation extends to environments in a natural way.

Definition 6.9 ($CE \models E : \Gamma$). $CE \models E : \Gamma$
if $Dom(E) = Dom(\Gamma)$,
and $CE \models E(x) : \Gamma(x)$ for any $x \in Dom(E)$

The definition below follows in the same vein and relates action sequences which annotate the evaluation rules of the dynamic semantics to the effects derived for expressions by the type system.

Definition 6.10 **($CE \models w : \kappa$).**

$CE \models \varepsilon : \kappa$	for any κ
$CE \models \mathsf{new}\ k : \kappa$	if $k \in Dom(CE)$ and $CE(k) = (\bar{\tau}, l)$ and $\{\mathtt{new}\, l \text{ for } \bar{\tau}\} \subseteq \kappa$
$CE \models k!v : \kappa$	if $k \in Dom(CE)$ and $CE(k) = (\bar{\tau}, l)$ and $CE \models v : (\bar{\tau}, l)$ and $\{\mathsf{send}\, \bar{\tau}\, \mathsf{on}\, l\} \subseteq \kappa$
$CE \models k?v : \kappa$	if $k \in Dom(CE)$ and $CE(k) = (\bar{\tau}, l)$ and $CE \models v : (\bar{\tau}, l)$ and $\{\mathsf{recv}\, \bar{\tau}\, \mathsf{on}\, l\} \subseteq \kappa$
$CE \models w_1.w_2 : \kappa \cup \kappa'$	if $CE \models w_1 : \kappa$ and $CE \models w_2 : \kappa'$

The consistency theorem stated below assumes a closed system composed of two expressions e_1 and e_2 which are evaluated at sites s_1 and s_2 respectively. Expressions are assumed to be well-typed and the initial evaluation environment at each site is assumed to be consistent with the initial static environment in the sense defined above. The theorem says that under these assumptions, if the evaluation at a site terminates yielding a value then this value must be related to the type derived for the expression.

Theorem 6.1 (Consistency). Assume the following

- $E_1 \vdash_{s_1} e_1 \stackrel{w_1}{\Longrightarrow} v_1$ and $E_2 \vdash_{s_2} e_2 \stackrel{w_2}{\Longrightarrow} v_2$ and $IOmatch(w_1, w_2)$
- $\Gamma_1 \vdash_{s_1} e_1 : \tau_1, \kappa_1$ and $\Gamma_2 \vdash_{s_2} e_2 : \tau_2, \kappa_2$
- $CE \models E_1 : \Gamma_1$ and $CE \models E_2 : \Gamma_2$

then there exists a CE' such that

- $CE \sqsubseteq CE'$
- $CE' \models v_1 : \tau_1$ and $CE' \models w_1 : \kappa_1$
- $CE' \models v_2 : \tau_2$ and $CE' \models w_2 : \kappa_2$

Proof. The proof is given by induction on the depth of the evaluation tree. For those cases which involve no communication we give the proof by considering e_1 only. It should be noted that the proof would be similar for expression e_2.

cases (con), (id), (fn). The required result follows from the assumptions and from Definitions 6.8, 6.9 and 6.10.

case $e_1 = \mathsf{chan}^l()$. The evaluation at site s_1 must have followed the rule (chan) such that $E_1 \vdash \mathsf{chan}() \stackrel{\mathsf{new}\, k}{\Longrightarrow} k$. Suppose that $\Gamma_1 \vdash \mathsf{chan}^l() : (\mathsf{chan}[\bar{\tau}], l)$. We take a CE' such that $CE' = CE[k \mapsto (\bar{\tau}, l)]$. The required result is immediate by Definition 6.8.

case $e_1 = (\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2)$. The evaluation must have followed the rule (let) and must be of the form $\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \stackrel{w_1.w_2}{\Longrightarrow} v_2$ where $E \vdash e_1 \stackrel{w_1}{\Longrightarrow} v_1$ and $E[x \mapsto v_1] \vdash e_2 \stackrel{w_2}{\Longrightarrow} v_2$. Suppose that $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2, \kappa \cup \kappa'$ where $\Gamma \vdash e_1 : \tau, \kappa$ and $\Gamma[x \mapsto Gen(\Gamma, \kappa, \tau)] \vdash e_2 : \tau_2, \kappa'$.

By the induction hypothesis there exists a CE' such that $CE \sqsubseteq CE'$ and $CE' \models v_1 : \tau_1$ and $CE' \models w_1 : \kappa$. If we could show that $CE' \models v_1 : Gen(\Gamma, \kappa, \tau_1)$ then we could apply the induction hypothesis on e_2. Let $Gen(\Gamma, \kappa, \tau_1) = \forall \vec{\gamma}.\tau_1$. By Definition 6.8 this boils down to showing that $CE' \models v_1 : \theta\tau_1$ for θ defined on $\vec{\gamma}$. If v_1 is a constant then there is only a unique type for v_1, hence the result is immediate. If v_1 is a channel identifier, $\mathsf{new}\, k$ must be a part of w_1 . Since $CE' \models k : (\mathsf{chan}[\bar{\tau}], l)$, according to Definition 6.8 $CE'(k) = (\bar{\tau}, l)$. Since $CE' \models w_1 : \kappa$, according to Definition 6.10, it must be the case that $\{\mathtt{new}\, l \text{ for } \bar{\tau}\} \subseteq \kappa$. We know by the definition of Gen that $\vec{\gamma}$ do not occur free in Γ or κ. Therefore, $CE' \models v_1 : \theta\tau_1$ because θ has no effect on τ_1. If v_1 is a closure the proof follows a similar idea. Additionally, it refers to the fact that if $\Gamma \vdash e : \tau, \kappa$ then $\theta\Gamma \vdash e : \theta\tau, \theta\kappa$. This lemma can be proved by following a similar reasoning to the various type substitution lemmas presented in the preceding chapters. Having proved that $CE' \models v_1 : Gen(\Gamma, \kappa, \tau_1)$ we can deduce that $CE' \models E[x \mapsto v_1] : \Gamma[x \mapsto \tau_1]$. We can now apply the induction hypothesis on e_1 to establish the required result.

cases (send) (receive). The proofs of the cases which involve communication make use of the hypothesis $IOmatch(w_1, w_2)$. This predicate states that for each send there is a corresponding receive and for each receive there is a send at the remote site. A channel environment which binds the channel identifier used in the communication to the type of the sent value should be taken as the channel environment which conforms to the requirements. The rest of the proof is rather straightforward. □

6.4.2 Noninterference

At various points in our discussions we stated that the dependency of public outputs of a system on secret inputs is undesirable and should be prevented. This is the same as requiring that the changes in the secret inputs to the system do not lead to changes on the public outputs. The noninterference theorem is a formalization of this statement. As a first step, we make precise when two public values are considered to be equivalent by means of the following definition.

Definition 6.11 (Equivalence of public values and environments). The notion of equivalence for public constants and channels is obvious. However, in principle it is impossible to decide the equality of two functions. We adopt a notion of equivalence which is based on the syntactic equality of function bodies. Two function closures are equivalent if the codes they enclose are identical and the environments they enclose are equivalent.

$$
\begin{array}{ll}
c^l \equiv_L c^l & \text{if } l = L \\
k \equiv_L k & \text{if } k = (L, s, i) \\
\langle l, E, x, e \rangle \equiv_L \langle l, E', x, e \rangle & \text{if } l = L \text{ and } E \equiv_L E' \\
& \\
E \equiv_L E' & \text{if } E(x) = E'(x) \text{ for all } x \text{ such that} \\
& x \in Dom(E) \text{ and } x \in Dom(E') \\
& \text{and } Label(E(x)) = Label(E'(x)) = L.
\end{array}
$$

Values can flow into and out of concurrently executing expressions by means of communication on channels. In a deterministic system consisting of two threads of control such as ours, the source of an incoming value is always known. The received value can only have been sent by the expression evaluated at the remote site. We enforce noninterference by ensuring that the values sent on public channels by an expression do not depend on the changes in the secret values which flow into it. Note that this also ensures that values which flow into the expression at the other site on communication channels remain the same.

The noninterference theorem makes use of the action sequences which instrument the evaluation rules to formalize the constraint on the public values which are sent by expressions.

Definition 6.12 (Purging secret send actions). The operation *Purge* is defined on action sequences. It purges all but send actions on public channels from a sequence.

$$Purge(\varepsilon) = \varepsilon$$
$$Purge(\mathsf{new}\ k.w) = Purge(w)$$
$$Purge(k\,!\,v.w) = \begin{cases} Purge(w) & \text{if } Label(k) = H \\ Purge(k\,!\,v.w) & \text{otherwise} \end{cases}$$
$$Purge(k\,?\,v.w) = Purge(w)$$

Definition 6.13 (Equivalence of purged sequences). Two purged sequences are equivalent if there is a one-to-one correspondence between the send actions in each sequence up to the equivalence of the values transmitted.

$$\varepsilon \equiv_L \varepsilon$$
$$k!v_1.w_1 \equiv_L k!v_2.w_2 \quad \text{if } v_1 \equiv_L v_2 \text{ and } w_1 \equiv_L w_2.$$

The theorem below states that if we evaluate a well-typed expression at a given site against two different initial environments which are the same except at bindings for secret values and both evaluations terminate yielding a public value, these values are guaranteed to be equal. Moreover, the two evaluations agree on the values transmitted on public channels.

Theorem 6.2 (Noninterference). Suppose that the following hold for a typing environment Γ, evaluation environments E_1,E_2, E_1' and E_2', and a channel environment CE.

- $CE \models E_1 : \Gamma$ and $CE \models E_2 : \Gamma$
- $\Gamma \vdash e_1 : (\bar{\tau}_1, l_1), \kappa_1$ and $\Gamma \vdash e_2 : (\bar{\tau}_2, l_2), \kappa_2$
- $E_1 \equiv_L E_1'$ and $E_2 \equiv_L E_2'$
- $E_1 \vdash e_1 \stackrel{w_1}{\Longrightarrow} v_1$ and $E_2 \vdash e_2 \stackrel{w_2}{\Longrightarrow} v_2$ and $IOmatch(w_1, w_2)$

- $E_1' \vdash e_1 \stackrel{w_1'}{\Longrightarrow} v_1'$ and $E_2' \vdash e_2 \stackrel{w_2'}{\Longrightarrow} v_2'$ and $IOmatch(w_1', w_2')$

then

- if $l_i = L$ then $v_i \equiv v_i{}'$
- $Purge(w_i) \equiv_L Purge(w_i')$

Proof. The proof is given by induction on the depth of the evaluation evaluation tree. We give the proof case involving conditional expressions below. We consider the evaluation of expression e_1 only, the proof for e_2 would be similar.

case $e = (\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3)$. Suppose that $\Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : (\bar{\tau}, l'), \kappa \cup \kappa' \cup \kappa''$ where $\Gamma \vdash e_1 : (\mathsf{bool}, l), \kappa$ and $\Gamma \vdash e_2 : (\bar{\tau}, l'), \kappa' \quad \Gamma \vdash e_3 : (\bar{\tau}, l'), \kappa''$ and $l \sqsubseteq l'$ and $Safe(\kappa', l)$ and $Safe(\kappa'', l)$.

For the first part of the proof we need to show that if $l' = L$ then $v_1 \equiv_L v_1'$. If $l' = L$ then $l = L$. Otherwise, the side condition $l \sqsubseteq l'$ would not be true. By applying the induction hypothesis on e_1 we know that if $E_1 \vdash e_1 \stackrel{w_3}{\Longrightarrow} \mathsf{true}^L$ then $E_1' \vdash e_1 \stackrel{w_3'}{\Longrightarrow} \mathsf{true}^L$. Similarly if $E_1 \vdash e_1 \stackrel{w_4}{\Longrightarrow} \mathsf{false}^L$ then $E_1' \vdash e_1 \stackrel{w_4'}{\Longrightarrow} \mathsf{false}^L$. This shows that the evaluation takes the same branch regardless of the value of the guard. That is to say if the resulting value v_1 has been obtained by following the evaluation rule (if-t) such that $E_1 \vdash e_2 \stackrel{w_5}{\Longrightarrow} v_1$ then the value v_1' will also be obtained by following the same rule such that $E_1{}' \vdash e_2 \stackrel{w_5'}{\Longrightarrow} v_1'$. By applying the induction hypothesis on e_2 we establish the required result; that is $v_1 \equiv_L v_1'$. Note that the same line of reasoning would be applicable if e_1 evaluated to false^L instead.

For the second part of the proof we need to show that $Purge(w_1) = Purge(w_1')$. If $l' = L$ and the rule (if-t) has been applied the proof follows by applying the induction hypothesis on e_1 and e_2. We have assumed that $w_1 = w_3.w_5$. By the induction hypothesis we know that $Purge(w_3) \equiv_L Purge(w_3')$ and $Purge(w_5) \equiv_L Purge(w_5')$. It is easy to check that $Purge(w_3.w_5) \equiv_L Purge(w_3', w_5')$. The same line of reasoning would be applicable if the evaluation was assumed to have followed the rule (if-f).

If $l' = H$ we can no longer know that the evaluation takes the same branch in the two independent runs, against the environments E_1 and E_1' respectively. This is because the application of the induction hypothesis does not say anything about the value of the guard which determines the branch taken. Let $w_1 = w.w'$ where $E_1 \vdash e_1 \stackrel{w}{\Longrightarrow} v$ and ${w_1}' = w''.w'''$ where ${E_1}' \vdash e_1 \stackrel{w''}{\Longrightarrow} v'$. All we can know by the induction hypothesis on e_1 is that $Purge(w) \equiv_L Purge(w'')$. If we can show that $Purge(w') \equiv_L Purge(w''')$ we can establish the required result $Purge(w.w') \equiv_L Purge(w''.w''')$.

By the typing rule (if) and Theorem 6.1 we know that for some CE', $CE' \models w' : \kappa'$ and $CE' \models w'' : \kappa''$ for some CE'. Since $Safe(\kappa', l)$ and $Safe(\kappa'', l)$ we know that neither κ' nor κ'' contains an element of the form $\mathtt{send}\,\bar{\tau}\,\mathtt{on}\,L$. By referring to Definition 6.10 we can deduce that neither w' nor w'' can have as a subsequence an action of the form $k!v$ where $Label(k) = L$. This implies that the application of the operation *Purge* on w' and w''' yields empty sequences. Since $Purge(w) \equiv_L Purge(w'')$ we can state that $Purge(w.w') \equiv_L Purge(w''.w''')$.

$\square$

6.5 Concluding remarks

In this chapter we have introduced the concept of noninterference and pointed out its significance in formulating secure information flow properties for systems and programs. We have then focused on a functional language which supports the mobility of functions between remote sites. In line with the general theme of this thesis, we investigated the applicability of type and effect systems in enforcing a noninterference property. The related work section of the previous chapter contains several pointers to works by other authors on type systems for security. Many of these works address noninterference in a given framework. The SLam [HR98a] calculus which is already discussed in detail in the previous chapter is closely related to Secure Mobile-λ as well as Confined-λ. The authors of SLam investigate static certification of programs which satisfy a lattice-based information flow property as presented in [Den76]. To simplify the discussion they restrict to a two-point lattice and note that the results can be generalized to any lattice of security classes. This is also the approach we have adopted. The results of this chapter can be generalized to any security lattice.

Nondeterminism This chapter considers a simple system which consists of two remote sites each of which has a trusted Secure Mobile-λ compiler. The noninterference property which holds for all well-typed programs of the language Secure Mobile-λ relies on the computation being deterministic. We have restricted the system to be composed of two sites where each site hosts a single thread. Since there is no shared mutable state and the communication can only take place between two parties, the concurrency does not give rise to nondeterminism.

This setting is sufficiently general to model the execution of single threaded mobile code by a host program which is also single threaded. The approach to security adopted in this chapter is in line with the work on a Web browser with applets written in CAML, a strongly typed functional language of the ML family [LR98].

The generalization of noninterference to nondeterministic systems gives rise to different characterizations of noninterference which are out of the scope of our work. In a nondeterministic concurrent setting, it would be acceptable for a system to output different public values in two different runs. Noninterference should then require that the values *possible* public outputs do not depend on the inputs. This notion of noninterference is referred to as *possibilistic noninterference*. However, possibilistic noninterference becomes inadequate when one considers the implementation of concurrent programs. Programs which satisfy possibilistic noninterference can still leak information by probabilistic inference. If one knows, for example, the probability of a thread being scheduled, the observation of the outputs can reveal information about the high-level inputs to the programs. *Probabilistic noninterference* rectifies this problem by requiring the probability distribution of the public outputs to be independent of the high-level inputs.

The work presented in this chapter constitutes a first step in the direction of establishing a robust and general notion for noninterference for mobile computation with functions where computation need not be deterministic. We have explored the applicability of the type and effect discipline in tracing information flow in a higher-order functional language with channel-based communication in a simple setting. This work could be taken further by considering possibilistic and probabilistic noninterference for a more general setting.

Blocked communications In languages which adopt synchronous communication such as Secure Mobile-λ the ability to send on a channel implies the existence of a receiver who has received the sent value. Otherwise, the send operation would have blocked. This makes languages such as Secure Mobile-λ vulnerable to information leaks which would not arise if the communication were asynchronous. The example below shows two expressions, e_1 and e_2 which are executed in parallel.

$$\begin{aligned} e_1 &= \mathsf{if\ secretBool} \quad \mathsf{then}\ (\mathsf{publicChannel?}; \mathsf{true}^H) \\ &\qquad\qquad\qquad\quad \mathsf{else\ false}^H \\ e_2 &= \mathsf{publicChannel!}()^L \end{aligned}$$

The expression e_2 attempts to leak the value of $\mathsf{secretBool}$ by attempting to send a value on the public channel $\mathsf{publicChannel}$. If the communication succeeds this would imply that $\mathsf{secretBool}$ is true.

The type system of this chapter does not aim at eliminating these kinds of information leaks. We have included this example here only to emphasize that in computational models such as that of Secure Mobile-λ blocked communications constitute a source of undesirable information flow.

Chapter 7

Conclusions

Each technical chapter in this thesis focuses on a practically-motivated problem concerning mobile computation in modern distributed systems. These problems arise from the heterogeneity of distributed systems in terms of the nature of computing devices, security requirements of the information flowing in the system and the trust level of users. The idea which is emphasized throughout is that principled language design and the exploitation of static program analysis techniques can offer satisfactory solutions to the problems considered in this thesis.

7.1 Natural support for code mobility

We regard simplicity and the existence of a formal definition as highly desirable properties for a mobile code language. It is through these properties that one can make reliable predictions about the dynamic behaviour of mobile code by using informal or formal techniques.

Our survey of the languages Concurrent ML, Facile and PLAN has convinced us that endowing functions with first-class status and incorporating a language construct for their communication between remote sites constitutes a simple approach to deriving a mobile code language. Consequently, we have used this approach in defining a series of higher-order functional languages which support code mobility.

An appealing aspect of these languages is that programmers can maintain a view of programs as a collection of function definitions and computation as a series of local or remote applications of these functions. Moreover, with appropriate adaptations, the existing program analysis techniques for functional programs continue to be useful. We demonstrated this by designing type and effect systems for the languages we defined and exploiting them to make predictions about several issues.

7.2 Type systems and security

Preventing the corruption of resources in a distributed system is a particularly challenging problem in the presence of code mobility. Memory is an essential resource for computation and type safety provides a basic protection mechanism against the corruption of memory. One of our goals has been to ensure that extensions for code mobility do not compromise type safety. The large body of work on type systems for concurrent functional languages assisted us in meeting this goal.

It is apparent that preventing the execution of those programs which may breach type safety does not suffice to protect systems against all security violations. This thesis presents several example programs which would be well-typed with respect to a conventional type system yet pose a threat to the availability of the resources or leak confidential information.

A significant part of the work presented in this thesis aims at extending the range of security properties which can be enforced by using type systems. The definition of security properties such as confinement in a mobility region and noninterference for mobile functions are our major contributions to the research area of foundations of security. It is widely acknowledged in this area that programming languages which support the development of code with provable security properties are essential for improving the security of systems in general. The simple languages presented in this work should be seen as prototypes of "secure" programming languages which can be of practical use in some application domains.

7.3 Further work

We concluded each chapter with a discussion of possible directions that could be followed to extend the work presented in that chapter. We now look at the thesis from a broader perspective and enumerate some ideas which apply to it on the whole.

The thesis presents a collection of statically typed languages and emphasizes the advantages of strong typing. We formulate and prove properties regarding the soundness of these type systems. The automatic inference of types has not been investigated here. There is a large body of work on this topic which could provide a starting point. It would be useful to devise algorithms to automate type inference and study their complexity and efficiency in practice. If the results are found to be promising, they will strengthen the arguments of this thesis.

Our approach has been to lay out a problem faced by mobile code languages and to discuss how types can gather useful information about programs with respect to the problem under investigation. In some cases, our attempts were limited to proposing a way to gather this information rather than showing how to use it in building tools or transforming programs. It would be interesting, for example, to present some optimizations based on the estimation of mobile values or to build tools for estimating resource-consumption.

Throughout we have assumed closed systems and relied on trusted parties to type-check all the code in the system. Although this is reasonable for many distributed computing environments, for a more general applicability of our work it is desirable to consider computing environments where global-type checking is impossible. This prompts us to study type systems which may resort to dynamic checks where the static checks turn out to be inadequate.

The constant developments in the area of mobile computation, both in theory and practice, are likely to give rise to many more directions for further research on the topic of this thesis.

Appendix A

Selected Proof Cases

A.1 Selected proof cases from Chapter 3

Lemma 3.1 If $\Gamma \vdash ie : \tau, \kappa$ then $\theta\Gamma \vdash ie : \theta\tau, \theta\kappa$ for any substitution θ.

Proof. The proof is given by induction on the derivation of $\Gamma \vdash ie : \tau, \kappa$.
case $ie = x$. The rule (id) of the the static semantics requires the typing to be of the form $\Gamma \vdash x : \tau, \emptyset$ where $\Gamma(x) = \sigma$ and $\sigma \succ \tau$.

We perform the necessary renaming on bound variables of σ such that $\sigma = \forall\vec{\delta}.\tau_x$ and $\vec{\delta}$ is out of reach of θ. Let θ' be a substitution over $\vec{\delta}$ such that $\theta'\tau_x = \tau$. Now define a substitution θ'' with domain $\vec{\delta_i}$ by $\theta''(\delta_i) = \theta(\theta'(\delta_i))$. We then have

$$\theta''(\theta(\delta_i)) = \theta''(\delta_i) = \theta(\theta'(\delta_i)) \text{ for all } i\text{, since } \vec{\delta} \text{ are out of reach of } \theta$$
$$\theta''(\theta(\rho)) = \theta(\rho) = \theta(\theta'(\rho)) \text{ for all } \rho \text{ not in } \vec{\delta_i}\text{, since } \theta'' \text{ is defined on } \delta_i$$

Hence, $\theta''(\theta(\tau_x)) = \theta(\theta'(\tau_x)) = \theta(\tau)$ which shows that $\theta\tau$ is an instance of $(\theta\Gamma)(x)$. Therefore, $\theta\Gamma \vdash x : \theta\tau, \emptyset$.

case $ie = (\mathsf{let}\, x = ie_1 \,\mathsf{in}\, e_2)$. The rule (let) of the static semantics requires that $\Gamma \vdash ie_1 : \tau, \kappa$ and $\Gamma[x \mapsto Gen(\Gamma, \kappa, \tau)] \vdash e_2 : \tau', \kappa'$. Let $\forall\vec{\delta}.\tau$ be $Gen(\Gamma, \kappa, \tau)$. For any substitution θ, let us consider fresh $\vec{\delta}'$ and define θ' as the extension of $\theta_{\vec{\delta}}$ with $\theta\{\vec{\delta} \mapsto \vec{\delta}'\}$ where $\theta_{\vec{\delta}}$ represents the restriction of θ obtained by removing $\vec{\delta}$ from the domain of θ. By the definition of *Gen* we know that $\delta_i \in \vec{\delta}$ are not free in Γ or κ, otherwise they would not have been generalized. Therefore, $\theta\Gamma = \theta'\Gamma$ and $\theta\kappa = \theta'\kappa'$. By the definition of θ', $\vec{\delta}$ being out of reach, $\theta'(\forall\vec{\delta}.\tau) = \forall\vec{\delta}'.\theta'\tau$. Thus, $\theta'(\Gamma[x \mapsto Gen(\Gamma,\kappa,\tau)]) = \theta'(\Gamma[x \mapsto \forall\vec{\delta}'.\theta'\tau]) = (\theta\Gamma)[x \mapsto Gen(\theta\Gamma,\theta\kappa,\theta'\tau)]$.

By using the induction hypothesis on ie_1 with θ' we get $\theta'\Gamma \vdash ie_1 : \theta'\tau, \theta'\kappa$. By the definition of θ', $\theta\Gamma \vdash ie_1 : \theta'\tau, \theta\kappa$. By using the induction hypothesis on e_2 we get $\theta(\Gamma[x \mapsto Gen(\Gamma,\kappa,\tau)]) \vdash e_2 : \theta\tau', \theta\kappa'$ which is equivalent to the judgement $(\theta\Gamma)[x \mapsto Gen(\theta\Gamma,\theta\kappa,\theta'\tau)] \vdash e_2 : \theta\tau', \theta\kappa'$. Finally by the typing rule (let) we can conclude that $\theta\Gamma \vdash \mathsf{let}\, x = ie_1 \,\mathsf{in}\, e_2 : \theta\tau', \theta\kappa'$.

$\square$

Theorem 3.3 If a type can be derived for an expression e in the type system then there exist an environment $\theta^p\Gamma$, a type τ^p and κ^p such that $\theta^p\Gamma \vdash e : \tau^p, \kappa^p$ and whenever $\theta\Gamma \vdash e : \tau, \kappa$ then for some substitution ψ it is the case that $\psi(\theta^p\Gamma) = \theta\Gamma$ and $\psi\tau^p = \tau$ and $\kappa \subseteq \psi\kappa^p$. The type τ^p is principal for e in Γ.

Proof. The proof is given by induction on depth of the typing derivation for e.
case $e = c$. The typing rule (con) for constants requires that $\theta\Gamma \vdash c : TypeOf(c), \emptyset$. By Definition 3.6, the type assigned to c is unique and is independent of the typing environment. We can take the substitution Id which maps all variables to themselves as the principal substitution required by the proof. Obviously, $\theta(Id\Gamma) = \theta\Gamma$ and $\theta(TypeOf(c)) = TypeOf(c)$ and $\emptyset \subseteq \theta\emptyset$. In this case $TypeOf(c)$ is the principal type and $\emptyset$ is the minimal effect.

case $e = x$. The typing rule (id) for identifiers requires that $(\theta\Gamma)(x) \succ \tau$ for any type τ which can be assigned to x. We take $\Gamma \vdash x : \tau^p, \emptyset$ as the principal typing for the variable x and show that whenever $\theta\Gamma \vdash x : \tau, \kappa$ then for some ψ it is the case that $\psi\Gamma = \theta\Gamma$ and $\psi\tau_p = \tau$ and $\emptyset \subseteq \psi\kappa$. The part of the proof involving the effects is trivial since $\emptyset$ is a subset of any set. For the part of the proof involving the type, let $\Gamma(x) = \forall\delta_1 \ldots \delta_n.\tau_x$ and $\theta'' = [\delta_1 \mapsto \iota_1, \ldots, \delta_n \mapsto \iota_n]$ for fresh ι_i and $\tau^p = \theta''\tau_x$ and $\tau = \theta'\tau_x$. A substitution ψ which is a composition of θ' and $[\iota_1 \mapsto \delta_1, \ldots, \iota_n \mapsto \delta_n]$, that is to say $\psi = \theta' \circ ([\iota_1 \mapsto \delta_1, \ldots, \iota_n \mapsto \delta_n])$ satisfies our condition that $\psi\tau^p = \tau$ and $\psi\Gamma = \theta\Gamma$.

case $e = (\mathsf{fn}^l\, x \Rightarrow e)$. The typing of a function expression such as this one follows the typing rule (fn) so that if $\theta\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e : (\tau \xrightarrow{\kappa} \tau', \{l\} \cup \mu), \emptyset$ it must be the case that $(\theta\Gamma)[x \mapsto \tau] \vdash e : \tau', \kappa$ and $\mu = M(\theta\Gamma, FI(\mathsf{fn}^l\, x \Rightarrow e))$. We can write $(\theta \circ [\alpha \mapsto \bar{\tau}, \gamma \mapsto \mu'])(\Gamma[x \mapsto (\alpha, \gamma)])$ for $(\theta\Gamma)[x \mapsto \tau]$ where α and γ are fresh and $\tau = (\bar{\tau}, \mu')$.

By applying the induction hypothesis on e we conclude that there exists a principal typing for e such that $(\theta^p\Gamma[x \mapsto (\alpha, \gamma)]) \vdash e : \tau'^p, \kappa^p$ and a substitution ψ such that $\psi(\theta^p(\Gamma[x \mapsto (\alpha, \gamma)])) = (\theta \circ [\alpha \mapsto \bar{\tau}, \gamma \mapsto \mu'])(\Gamma[x \mapsto (\alpha, \gamma)])$ and $\psi\tau'^p = \tau'$ and $\kappa^p \subseteq \psi\kappa$. Let $\mu^p = \{M(\theta^p\Gamma, FI(\mathsf{fn}^l\, x \Rightarrow e))$.

The equality $\psi(\theta^p(\Gamma[x \mapsto (\alpha, \gamma)])) = (\theta \circ [\alpha \mapsto \bar{\tau}, \gamma \mapsto \mu])(\Gamma[x \mapsto (\alpha, \gamma)])$ implies that $\psi(\theta^p\Gamma) = \theta\Gamma$ and $\psi(\alpha, \gamma) = \tau$. Let $\tau^p = (\alpha, \gamma)$ and take the principal typing as $\theta^p\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e : (\tau_p \xrightarrow{\beta \cup \kappa_p} {\tau_p}', \mu_p)$ where β is fresh. We know that $\psi\tau_p = \tau$ and $\kappa^p \subseteq \psi\kappa$. Since $\psi(\theta^p\Gamma) = \theta\Gamma$, by Definition 3.7 we can deduce $\psi\mu_p = \mu$. Let $\kappa'' = \kappa \setminus \psi\kappa_p$. The substitution $\psi' = \psi \circ [\beta \mapsto \kappa'']$ satisfies the necessary conditions.

case $e = (\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2)$ The typing of a let expression such as this one follows the typing rule (let) so that if $\theta\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau', \kappa'$, we can assume that $\theta\Gamma \vdash e_1 : \tau, \kappa$ and $\theta\Gamma[x \mapsto Gen(\theta\Gamma, \kappa, \tau)] \vdash e_2 : \tau', \kappa'$.

By applying the induction hypothesis on e_1 there exists a principal typing $\theta^p\Gamma \vdash e_1 : \tau^p, \kappa^p$ and a ψ such that $\psi(\theta\Gamma) = \theta^p\Gamma$ and $\psi\tau^p = \tau$ and $\psi\kappa^p \subseteq \kappa$. By referring to Definition 3.5, one can show that $\psi(Gen(\theta^p\Gamma, \kappa^p, \tau^p)) \succ Gen(\theta\Gamma, \kappa, \tau)$. Since $\psi(\theta\Gamma) = \theta\Gamma$, whenever $(\theta\Gamma)[x \mapsto Gen(\theta\Gamma, \kappa, \tau)] \vdash e_2 : \tau', \kappa'$ we also have the judgement $(\psi(\theta\Gamma))[x \mapsto \psi(Gen((\theta\Gamma, \kappa, \tau))] \vdash e_2 : \tau', \kappa'$. By applying the induction hypothesis on e_2 we have $\psi(\theta^p\Gamma)$ and $\psi\tau^{p'} = \tau'$ and $\psi\kappa^{p'} \subseteq \kappa'$. By applying the typing rule (let) we can conclude that $\theta^p\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau^{p'}, \kappa^{p'}$ is the principal typing.

□

A.2 Selected proof cases from Chapter 4

Theorem 4.1 Let e be a closed expression which is evaluated at site s in the system. Assume $S[(s,p) : e] \xrightarrow{a} S'[(s,p) : e']$ and $\Gamma \vdash e : \tau, \mathcal{F}$. Then $\Gamma \vdash e' : \tau, \mathcal{F}$ and $a \models \overline{\{\mathcal{F} @ \{s\}\}}$.

Proof. **case** $e = (\mathsf{rec}^l\, f\,(x) \Rightarrow e)$. The dynamic semantics requires the evaluation to follow the rule (rec) such that $S[(s,p) : (\mathsf{rec}^l\, f\,(x) \Rightarrow e)] \xrightarrow{\varepsilon} S[(s,p) : \mathsf{fn}^l\, x \Rightarrow e\{(\mathsf{rec}^l\, f\,(x) \Rightarrow e)/f\}]$. Suppose $\Gamma \vdash \mathsf{rec}^l\, f\,(x) \Rightarrow e : \tau \xrightarrow{\{l\}\cup\mathcal{F}} \tau', \mathcal{F}'$. By subsumption elimination we know that $\Gamma \vdash \mathsf{rec}^l\, f\,(x) \Rightarrow e : \tau \xrightarrow{\{l\}\cup\mathcal{F}} \tau', \emptyset$ and by the typing rule (rec) we know that $\Gamma[x \mapsto \tau][f \mapsto (\tau \xrightarrow{\{l\}\cup\mathcal{F}} \tau')] \vdash e : \tau', \{l\} \cup \mathcal{F}$. Since $\Gamma \vdash \mathsf{rec}^l\, f\,(x) \Rightarrow e : \tau \xrightarrow{\{l\}\cup\mathcal{F}} \tau', \emptyset$ and x is not free in the recursive expression we can state that $\Gamma[x \mapsto \tau] \vdash \mathsf{rec}^l\, f\,(x) \Rightarrow e : \tau \xrightarrow{\{l\}\cup\mathcal{F}} \tau', \emptyset$. This allows us to apply Lemma 4.1 to deduce $\Gamma[x \mapsto \tau] \vdash e\{(\mathsf{rec}^l\, f\,(x) \Rightarrow e)/f\} : \tau', \{l\} \cup \mathcal{F}$. By using the typing rule (fn) for functions we know that $\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e\{(\mathsf{rec}^l\, f\,(x) \Rightarrow e)/f\} : \tau \xrightarrow{\{l\}\cup\mathcal{F}} \tau', \emptyset$. It follows by the rule (subs) that $\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e\{(\mathsf{rec}^l\, f\,(x) \Rightarrow e)/f\} : \tau \xrightarrow{\{l\}\cup\mathcal{F}} \tau', \mathcal{F}'$. This concludes the proof of the first part. The proof of the second part is trivial since ε is estimated by any set according to Definition 4.2.

case $e = (\mathsf{reval}\,((\mathsf{fn}^l\, x \Rightarrow e), v)\, \mathsf{at}\, s')$. The dynamic semantics requires the evaluation to follow the rule (reval-4) such that $S[(s,p) : \mathsf{reval}\,((\mathsf{fn}^l\, x \Rightarrow e), v)\, \mathsf{at}\, s'] \xrightarrow{l@s'} S[(s,p) : \mathsf{block}_{(s',p')}\, \mathsf{on}\, e\{v/x\}][(s',p') : e\{v/x\}]$ where p' is new at s'. Suppose that we have $\Gamma \vdash \mathsf{reval}\,((\mathsf{fn}^l\, x \Rightarrow e), v)\, \mathsf{at}\, s' : \tau', \mathcal{F}$. By subsumption elimination we know that there exist $\mathcal{F}'$ and $\mathcal{S}$ such that $\overline{\{\mathcal{F}'@\mathcal{S}\}} \subseteq \overline{\mathcal{F}}$ and $\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e : \tau \xrightarrow{\mathcal{F}'} \tau', \emptyset$ and $\Gamma \vdash v : \tau, \emptyset$ and $\Gamma \vdash s' : \mathsf{site}^{\mathcal{S}}, \emptyset$. We are required to show that $\Gamma \vdash \mathsf{block}_{(s',p')}\, \mathsf{on}\, e\{v/x\} : \tau', \mathcal{F}$.

By the typing rule (fn) for functions we know that $\Gamma[x \mapsto \tau] \vdash e : \tau', \mathcal{F}''$ where $\{l\} \cup \mathcal{F}'' = \mathcal{F}'$ for some $\mathcal{F}''$ and by appealing to Lemma 4.1 it can be shown that $\Gamma \vdash e\{v/x\} : \tau', \mathcal{F}''$. The typing rule for blocked expressions and the typing rule (subs) allow us to conclude the first part of the proof. For the proof of the second part, we have to show that $l@s' \models \overline{\{\mathcal{F}@s\}}$. We know that $\mathcal{F}' = \{l\} \cup \mathcal{F}''$ for some $\mathcal{F}''$. By the typing rule for (site) it must be the case that $s' \in \mathcal{S}$. It is easy to show that $l@s' \models \overline{\{\mathcal{F}@s\}}$ by referring to Definitions 4.1 and 4.2. $\square$

Proposition 4.1 Let I be a non-empty set of indices and J be the set of possible typing judgements for an expression e defined as follows:

$$J = \{\Gamma^i \vdash e : \tau^i, \mathcal{F}^i \mid i \in I, \lfloor \Gamma^j \rfloor = \lfloor \Gamma^k \rfloor, \lfloor \tau^j \rfloor = \lfloor \tau^k \rfloor \text{ for all pairs } j,k \in I\}.$$

Then there exists a minimal element of J, written as $\sqcap\Gamma \vdash e : \sqcap\tau, \sqcap\mathcal{F}$, such that for all $i \in I$ it is the case that $\sqcap\Gamma \sqsubseteq \Gamma^i$ and $\sqcap\tau \sqsubseteq \tau^i$ and $\sqcap\mathcal{F} \sqsubseteq \mathcal{F}^i$.

Proof. **case** $e = x$. It is possible to find a greatest lower bound for any two types which have the same underlying type structure. Since all the annotations are in the form of sets, this can be achieved by taking the intersection of the annotations. The proof case for typing identifiers makes use of this fact. Suppose that $J = \{\Gamma^i \vdash x : \tau^i, \mathcal{F}^i\}$. Obviously, for all $\Gamma^i \vdash x : \tau^i, \mathcal{F}^i$, it is the case that $(\sqcap\Gamma^i) \sqsubseteq \Gamma^i, \tau^i \sqsubseteq (\sqcap\Gamma^i)(x)$ and $\emptyset \sqsubseteq \mathcal{F}^i$.

case $e = (\mathsf{fn}^l\, x \Rightarrow e')$. Suppose that $J = \{\Gamma^i \vdash \mathsf{fn}^l\, x \Rightarrow e' : {\tau_1}^i \xrightarrow{\{l\}\cup\mathcal{F}^i} {\tau_2}^i, \mathcal{F}^i \mid i \in I\}$. The typing rule (fn) for functions requires that for each element of J we have a corresponding judgement $\Gamma^i[x \mapsto {\tau_1}^i] \vdash e' : {\tau_2}^i, \mathcal{F}^i$. Let J' be the set of these judgements such that $J' = \{\Gamma^i[x \mapsto {\tau_1}^i] \vdash e' : {\tau_2}^i, \mathcal{F}^i \mid i \in I, \lfloor\Gamma_j[x \mapsto {\tau_1}^j]\rfloor = \lfloor\Gamma_k[x \mapsto {\tau_1}^k]\rfloor, \lfloor{\tau_2}^j\rfloor = \lfloor{\tau_2}^k\rfloor$ for all pairs $j,k \in I\}$. We apply the induction hypothesis to the set J' and this allows us to conclude that there exists a minimal element of J'. Let $(\sqcap\Gamma)[x \mapsto \sqcap\tau_1] \vdash e' : \sqcap\tau_2, \sqcap\mathcal{F}$ be this minimal element. By the typing rule (fn) we can deduce that $\sqcap\Gamma \vdash e' : \sqcap\tau_1 \xrightarrow{\{l\}\cup\sqcap\mathcal{F}} \sqcap\tau_2, \emptyset$ is a possible typing for the function. Since $\sqcap\mathcal{F} \sqsubseteq \mathcal{F}^i$ for all $i \in I$, it must be the case that $\mathit{Flatten}(\mathcal{F}) \subseteq \mathit{Flatten}(\mathcal{F}^i)$ for all $i \in I$. It follows that $\{l\} \cup \sqcap\mathcal{F} \sqsubseteq \{l\} \cup \mathcal{F}^i$ and $\sqcap\tau_1 \xrightarrow{\{l\}\cup\sqcap\mathcal{F}} \sqcap\tau_2 \sqsubseteq {\tau_1}^i \xrightarrow{\{l\}\cup\mathcal{F}^i} {\tau_2}^i$for all $i \in I$. Hence we have shown that a minimal element for J, the set of possible typing judgements for the function exists.

case $e = (\mathsf{reval}\,(e_1, e_2)\,\mathsf{at}\, e_3)$. This proof case follows a similar line of reasoning as the case above. Suppose that $\sqcap\mathcal{F}$ is the minimal annotation derived for the type of e_1 and that $\sqcap\mathcal{S}$ is the minimal annotation derived for e_3. The only part of the proof which may appear not to be obvious is the part which involves showing that $\sqcap\mathcal{F} @ \sqcap\mathcal{S} \sqsubseteq \mathcal{F}^i @ \mathcal{S}^i$ for all i which index the possible annotations derivable for e_1 and e_3. The required result can be established by referring to Definition 4.3. □

Theorem 4.2 Let e be a closed expression which is evaluated at site s. Assume $S[(s,p) : e] \xrightarrow{a} S'[(s,p) : e']$ and $\Gamma \vdash e : \tau, \mathcal{F}$. Then $\Gamma \vdash e' : \tau, \mathcal{F}'$ where $\overline{\overline{\mathcal{F}'}}$ is a subsequence of $\overline{\overline{\mathcal{F}}}$ and $a \models_p \overline{\overline{\mathcal{F} @ s}}$.

Proof. **case** $e = (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$. The dynamic semantics requires the evaluation to follow the rule (let-1) such that $S[(s,p) : \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2] \xrightarrow{a} S'[(s,p) : \mathsf{let}\, x = e_1'\ \mathsf{in}\, e_2]$ where $S'[(s,p) : e_1']$. Suppose that $\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\, e_2 : \tau', \mathcal{F} \rhd \mathcal{F}'$ where $\Gamma \vdash e_1 : \tau, \mathcal{F}$ and $\Gamma[x \mapsto Gen(\Gamma, \mathcal{F}, \tau)] \vdash e_2 : \tau', \mathcal{F}'$. By applying the induction hypothesis on e_1 we have $\Gamma \vdash e_1' : \tau, \mathcal{F}''$ such that $\overline{\overline{\mathcal{F}''}}$ is a subsequence of $\overline{\overline{\mathcal{F}}}$ and $a \models_p \overline{\overline{\{\mathcal{F}@s\}}}$. Let $\sigma = \forall\vec{\delta}.\tau = Gen(\Gamma, \mathcal{F}, \tau)$ for $\vec{\delta}$ which are not free in Γ and $\mathcal{F}$. Since $\overline{\overline{\mathcal{F}''}}$ is a subsequence of $\overline{\overline{\mathcal{F}}}$, variables in $\mathcal{F}''$ must be a subset of the variables in $\mathcal{F}$. Therefore, $Gen(\Gamma, \mathcal{F}'', \tau) \succ Gen(\Gamma, \mathcal{F}, \tau)$ and $\Gamma[x \mapsto Gen(\Gamma, \mathcal{F}, \tau)] \vdash e_2 : \tau', \mathcal{F}'$ implies that $\Gamma[x \mapsto Gen(\Gamma, \mathcal{F}'', \tau)] \vdash e_2 : \tau', \mathcal{F}'$. By the typing rule (let), $\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau', \mathcal{F}'' \rhd \mathcal{F}'$. It remains to show that $\overline{\overline{\mathcal{F}'' \rhd \mathcal{F}'}}$ is a subsequence of $\overline{\overline{\mathcal{F} \rhd \mathcal{F}'}}$ and that $a \models_p \overline{\overline{\{\mathcal{F}@s\}}}$. The proof of the latter part is immediate from the induction hypothesis. The proof of the former part can be established by referring to Definition 4.8.

case $e = (\mathsf{reval}\,((\mathsf{fn}^l\, x \Rightarrow e), v)\ \mathsf{at}\ s')$. The dynamic semantics requires the evaluation to follow the rule (reval-4) such that $S[(s,p) : \mathsf{reval}\,((\mathsf{fn}^l\, x \Rightarrow e), v)\ \mathsf{at}\ s'] \xrightarrow{l@s'} S[(s,p) : \mathsf{block}_{(s',p')}\ \mathsf{on}\, e\{v/x\}][(s',p') : e\{v/x\}]$ where p' new at s'. Suppose that $\Gamma \vdash \mathsf{reval}\,((\mathsf{fn}^l\, x \Rightarrow e), v)\ \mathsf{at}\ s' : \mathsf{unit}, \mathcal{F}@s'$ where $\Gamma \vdash \mathsf{fn}^l\, x \Rightarrow e : \tau \xrightarrow{l \rhd \mathcal{F}'} \tau', *$ such that $\mathcal{F} = l \rhd \mathcal{F}'$ and $\Gamma \vdash v : \tau, *$. By the typing rule for functions we know that $\Gamma[x \mapsto \tau] \vdash e : \tau', \mathcal{F}'$ and by Lemma 4.3 we know that $\Gamma \vdash e\{v/x\} : \tau', \mathcal{F}'$. The typing rule for blocked expressions allows us to conclude that $\Gamma \vdash \mathsf{block}_{(s',p')}\ \mathsf{on}\, e\{v/x\} : (l \rhd \mathcal{F}')@\{s'\}$. Now, it remains to show that $\overline{\overline{(l \rhd \mathcal{F}')@\{s'\})}}$ is a subsequence of $\overline{\overline{* \rhd * \rhd \mathcal{F}@s'}}$ and that $l@s' \models_p \overline{\overline{\mathcal{F}@\{s'\}}}$. We know that $\mathcal{F} = l \rhd \mathcal{F}'$. The required result follows from Definition 4.8.

□

A.3 Selected proof cases from Chapter 5

Lemma 5.1 If $r, CE, \Gamma[x \mapsto \tau_1] \vdash e : \tau_2$ and $r, CE, \Gamma \vdash v : \tau_1$ then $r, CE, \Gamma \vdash e\{v/x\} : \tau_2$.

Proof. The proof is given by induction on the typing derivation of $r, CE, \Gamma[x \mapsto \tau_1] \vdash e : \tau_2$. Some representative cases are as follows:
case $e = x$. Suppose that $r, CE, \Gamma[x \mapsto \tau_1] \vdash x' : \tau_2$.

i. $x \neq x'$. This implies that x' is in the domain of Γ such that $\Gamma(x') = \tau_2$ satisfying the necessary conditions of the typing rule (id). We know that $x'\{v/x\} = x'$. We can conclude by the typing rule (var) that $r, CE, \Gamma \vdash x' : \tau_2$.

ii. $x = x'$. This implies that $\tau_2 = \tau_1$. Since $x\{v/x\} = v$ we need to establish $r, CE, \Gamma \vdash v : \tau_2$. This is immediate from the assumptions.

case $e = (\mathsf{fn}^r\, y \Rightarrow e)$. Suppose that $r', CE, \Gamma[x \mapsto \tau_1] \vdash \mathsf{fn}^r\, y \Rightarrow e : (\tau_2 \rightarrow \tau_3, r)$ where $r, CE, \Gamma[x \mapsto \tau_1][y \mapsto \tau_2] \vdash e : \tau_3$ and $r' \subseteq r$ and $x \neq y$. By induction hypothesis on e we have $r, CE, \Gamma[y \mapsto \tau_2] \vdash e\{v/x\} : \tau_3$. We get the required result by applying the typing rule (fn). □

Lemma 5.2 If $r, CE, \Gamma \vdash v : (\bar{\tau}, r')$ then for any r'' such that $r'' \subseteq r'$ it is the case that $r'', CE, \Gamma \vdash v : (\bar{\tau}, r')$.

Proof. The proof is given by induction on the typing derivation of $r, CE, \Gamma \vdash v : (\bar{\tau}, r')$. **case** $e = c^{r'}$. Suppose $r, CE, \Gamma \vdash c^{r'} : (\bar{\tau}, r')$ where $Type_{CE}(c^{r'}) = (\bar{\tau}, r')$ and $r \subseteq r'$. For any $r'' \subseteq r'$ the side conditions would still hold for establishing $r'', CE, \Gamma \vdash c : (\bar{\tau}, r')$.

case (fn) Similar to the above case.

case (op) $r, CE, \Gamma \vdash v_1\ op\ v_2 : (\mathsf{int}, r')$ where $r, CE, \Gamma \vdash v_1 : (\mathsf{int}, r_1)$ and $r, CE, \Gamma \vdash v_2 : (\mathsf{int}, r_2)$ and $r' = r_1 \cap r_2$. By induction hypothesis on v_1 and v_2 we have that $r'', CE, \Gamma \vdash : v_1 : (\mathsf{int}, r_1)$ and $r'', CE, \Gamma \vdash v_2 : (\mathsf{int}, r_2)$. The required result follows from the typing rule (op).

□

Theorem 5.1 Assume $CI, P[s : e] \longrightarrow CI', P'[s : e']$ and $CI = Dom(CE)$ and $r, CE, [\,] \vdash e : \tau$. Then there exists a CE' such that $r, CE', [\,] \vdash e' : \tau$ and $CI' = Dom(CE')$.

Proof. We have outlined the proof in Section 5.5.1. We include here the proofs for some of the representative cases only.

case (let-2)$CI, P[s : \mathsf{let}\, x = v \,\mathsf{in}\, e_2] \longrightarrow CI, P[s : e_2\{v/x\}]$ and $r, CE, [] \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e_2 : \tau_2$ where $r, CE, [] \vdash v : \tau$ and $r, CE, [x \mapsto \tau] \vdash e_2 : \tau_2$. By Lemma 5.1 we establish the required result.

case (com) $CI, P[s_1 : k^r!v][s_2 : v] \longrightarrow CI', P'[s_1 : ()^r][s_2 : v]$ and $CI = Dom(CE)$ and $\{s_1\}, CE, \Gamma \vdash k^r!v : (\mathsf{unit}, r)$ and $\{s_2\}, CE, \Gamma \vdash k^r? : (\bar{\tau}, r)$. The typing rules for expressions at s_1 and s_2 require that $\{s_1\}, CE, \Gamma \vdash k^r : (\mathsf{chan}[\bar{\tau}], r)$ where $\{s_1\} \subseteq r$ and $\{s_1\}, CE, \Gamma \vdash v : (\bar{\tau}, r)$ and $\{s_2\}, CE, \Gamma \vdash k^r : (\mathsf{chan}[\bar{\tau}], r)$ where $\{s_2\} \subseteq r$. It is obvious that $\{s_1\}, CE, \Gamma \vdash ()^r : (\mathsf{unit}, r)$. By Lemma 2, using $\{s_1\}, CE, \Gamma \vdash v : (\bar{\tau}, r)$ and $\{s_2\} \subseteq r$ as assumptions, we can establish $\{s_1\}, CE, \Gamma \vdash v : (\bar{\tau}, r)$ which is the required result. □

Lemma 5.3 If $r, CE, \Gamma \vdash e : \tau$ then $r \subseteq L_s(\tau)$.

Proof. The proof is given by induction on the typing derivation of $r, CE, \Gamma \vdash e : \tau$. Some representative cases are as follows: **cases** (con),(var) and (fn) follow immediately from the side conditions imposed by the typing rules. **case** $e = e_1 e_2$. Suppose $r, CE, \Gamma \vdash e_1 e_2 : \tau$. The typing rule (app) requires that $r, CE, \Gamma \vdash e_1 : (\tau \rightarrow \tau_1, r')$ and $r, CE, \Gamma \vdash e_2 : \tau$. By the induction hypothesis on e_1 we have that $r \subseteq r'$. It follows from the well-formedness condition for function types that $r' \subseteq L_s(\tau_1)$. By transitivity of the subset relation $\subseteq$ we conclude that $r \subseteq L_s(\tau_1)$.

□

Lemma 5.4 If $r, CE, \Gamma \vdash v : \tau$ then $L_d(v) = L_s(\tau)$.

Proof. A simple inspection of the typing rules is sufficient to prove this lemma. □

Bibliography

[AAH+99] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):37–45, 1999.

[Aba99] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[ABHR99] M. Abadi, A. Banarjee, N. Heintze, and J.G. Riecke. A core calculus of dependency. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 147–160. ACM Press, 1999.

[AFG98] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 105–116, 1998.

[AFG99] M. Abadi, C. Fournet, and G. Gonthier. Secure communications processing for distributed languages. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 74–88, 1999.

[AG99] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[Aga00] J. Agat. Transforming out time leaks. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 40–53, Boston, Massachusetts, USA, 2000.

[AM91] A. Appel and D.B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, volume 528 of *LNCS*, pages 1–26. Springer-Verlag, 1991.

[Ama00] R.M. Amadio. On modelling mobility. *Theoretical Computer Science*, 240:147–176, 2000.

[AP94] R.M. Amadio and S. Prasad. Localities and failures. In *Proceedings of 14th FST and TCS Conferences*, number 880 in LNCS, pages 205–216. Springer-Verlag, 1994.

[AWWR93] J. Armstrong, M. Williams, C. Wikstrom, and R.Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[BC01] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 222–235. ACM Press, 2001.

[BD97] D. Boloignano and M. Debbabi. A semantic theory for ML higher order concurrency primitives. In F. Nielson, editor, *ML with Concurrency*, chapter 6, pages 145–183. Springer-Verlag New York, Inc, 1997.

[BG92] G. Berry and G.Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[BMT92] D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 119–129. ACM Press, 1992.

[Bou97] G. Boudol. The pi-calculus in direct style. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241. ACM Press, 1997.

[Car99] L. Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*, pages 51–94. Springer, 1999.

[CG98] L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, number 1378 in LNCS, pages 140–155. Springer-Verlag, 1998.

[CG99] L. Cardelli and A.D. Gordon. Types for mobile ambients. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 79–92. ACM Press, 1999.

[CGG99a] L. Cardelli, G. Ghelli, and A.D. Gordon. Ambient groups and mobility types. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *LNCS*, pages 230–239. Springer Verlag, 1999.

[CGG99b] L. Cardelli, G. Ghelli, and A.D. Gordon. Mobility types for mobile ambients. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *LNCS*, pages 230–239. Springer Verlag, 1999.

[CGG00] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, volume 1877 of *LNCS*, pages 365–379. Springer-Verlag, 2000.

[CV99] G. Castagna and J. Vitek. Seal: A framework for secure mobile computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, number 1686 in Lecture Notes of Computer Science, pages 47–77. Springer-Verlag, 1999.

[CW00] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 184–198. ACM Press, 2000.

[Den76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[DJG92] V. Dornic, P. Jouvelot, and D. K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, March 1992.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212. ACM Press, 1982.

[FG95] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[FG96] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 372–385. ACM Press, 1996.

[FGMR96] C. Fournet, G. Gonthier, L. Maranget, and D. Remy. A calculus of mobile agents. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, volume 1119 of *LNCS*, pages 406–421, 1996.

[FHJ96] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 201–212. ACM Press, 1996.

[FLMR97] C. Fournet, C. Laneve, L. Maranget, and D. Remy. Implicit typing à la ML for the join calculus. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, volume 1243 of *LNCS*, pages 196–212, 1997.

[FM97] C. Fournet and L. Maranget. *The Join-Calculus Language release 1.04*. Institut National de Recherche en Informatique et Automatique, 1997. Available from `http://pauillac.inria.fr/join/`.

[FPV98] A. Fugetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[GFH97] K.L. Gasser, F.Nielson, and H.R.Nielson. Systematic realisation of control flow analysis for CML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 38–51. ACM Press, 1997.

[GM82] J. Gougen and Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[Hal85] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Hei94a] N. Heintze. Control-flow analysis and type systems. Technical Report CMU-CS-94-227, School of Computer Science, Carnegie Mellon University, 1994. Also appears as Fox Memorandum CMU-CS-FOX-94-09.

[Hei94b] N. Heintze. Set-based analysis of ML programs. In *Proceedings of Lisp and Functional Programming*. ACM Press, 1994.

[Hei95] N. Heintze. Control-flow and type systems. In *Proceedings of the Static Analysis Symposium (SAS)*, volume 983, pages 189–206. Springer-Verlag, 1995.

[Hen98] M. Hennessy. A survey of location calculi. Proceedings of the Fifth International Workshop on Expressiveness in Concurrency, 1998. Invited Talk.

[HK99] M. Hicks and A. D. Keromytis. A secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Workshop on Active Networks*, volume 1653 of *LNCS*, pages 307–314. Springer-Verlag, 1999.

[HKM+98] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 86–93. ACM Press, 1998.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[Hof00] M. Hofmann. A type system for bounded space and functional in-place update. In *Proceedings of the European Symposium on Programming (ESOP)*, volume 1782 of *LNCS*, pages 165–179. Springer-Verlag, 2000.

[HP99] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 70–81. ACM Press, 1999.

[HR98a] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, CA, USA, 1998.

[HR98b] M. Hennesy and J. Riely. Resource access control in systems of mobile agents. In *Proceedings of the 3rd International Workshop on High-Level Concurrent Languages (HLCL)*, volume 3, 1998. To appear in Information and Computation.

[HR98c] M. Hennesy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Proceedings of the 4th Workshop on Mobile Object Systems*, volume 1603 of *LNCS*, pages 95–117, Brussels, 1998. Springer-Verlag.

[HR99] M. Hennesy and J. Riely. Trust and partial typing in open systems of mobile agents. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 93–104. ACM Press, 1999.

[HR00] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 415–427, Geneva, Switzerland, 2000. Springer-Verlag.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 133–147, 1991.

[HVY00] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 180–199. Springer-Verlag, 2000.

[HY00] M. Hashimoto and A. Yonezawa. MobileML: A programming language for mobile computation. In *Proceedings of the 4th International Conference on Coordination Languages and Models (COORDINATION)*, volume 1906 of *LNCS*, pages 198–215. Springer-Verlag, 2000.

[Jef96] A. Jeffrey. Semantics for concurrent ML using computation types. Technical Report 96:03, School of Computing and Cognitive Sciences, University of Sussex, 1996.

[JGF96] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 295–308, Florida, January 1996.

[JLM97] S.P. Jones, J. Launchbury, and E. Meijer. Henk: a typed intermediate language. In *Proceedings of the Types in Compilation Workshop (TIC)*, 1997.

[JW95] S. Jaganathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 393–407. ACM Press, 1995.

[KGA00] P. Kakkar, C. A. Gunter, and M. Abadi. Reasoning About Secrecy for Active Networks. In *Proceedings of the Computer Security Foundations Workshop (CSFW), Electronic Edition*, pages 118–129. IEEE Computer Society DL, 2000.

[KHMG99] P. Kakkar, M. Hicks, J. T. Moore, and Carl A. Gunter. Specifying the PLAN network programming language. In *Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[Kır99a] Z. D. Kırlı. A static type system for detecting potentially transmissible functions. In P. Sewell and J. Vitek, editors, *Proceedings of the 5th Mobile Object Systems Workshop: Programming Languages for Wide Area Networks*, Lisbon, Portugal, 1999.

[Kır99b] Z. D. Kırlı. A survey on functions, concurrency, distribution and mobility. In *Preliminary Proceedings of the 1st Scottish Functional Programming Workshop*, Stirling,UK, 1999.

[Kır00] Z. D. Kırlı. Secure information flow for mobile functions. In *Proceedings of the Workshop on Issues in the Theory of Security*, Geneva, Switzerland, 2000.

[Kır01] Z. D. Kırlı. Confined mobile functions. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 283–295, Nova Scotia, Canada, 2001. IEEE Computer Society Press.

[Kna95] F.C. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, December 1995.

[Ler92] X. Leroy. *Polymorphic Typing of an Algorithmic Language*. PhD thesis, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France, 1992.

[Ler97] X. Leroy. Objective CAML, 1997. `http://pauillac.inria.fr/ocaml/`.

[LR98] X. Leroy and F. Rouaix. Security properties of typed applets. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 391–400. ACM Press, 1998.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 352–364. ACM Press, 2000.

[LT95] L. Leth and B. Thomsen. Some Facile chemistry. *Formal Aspects of Computing*, 7(E):67–110, 1995.

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, 1997.

[Mat97] D. Matthews. Concurreny in Poly/ML. In F. Nielson, editor, *ML with Concurrency*, chapter 3, pages 31–58. Springer-Verlag New York, Inc, 1997. Also available as a University of Edinburgh Technical Report ECS-LFCS-91-174.

[McL94] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley Press, 1994.

[MHN01] J. T. Moore, M. Hicks, and S. Nettles. Practical programmable packets. In *Proceedings of the Twentieth IEEE Computer and Communication Society INFOCOM Conference*. IEEE, April 2001.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[ML99] A. C. Myers and B. Liskov. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[MM94] P. Mosses and Musicante M. An action semantics for ML concurrency primitives. In *Proceedings of FME '94*, volume 873 of *LNCS*, pages 461–479. Springer-Verlag, 1994.

[Mor99] A. F. Moreira. *A Type Based Locality Analysis for a Functional Distributed Language*. PhD thesis, University of Edinburgh, 1999.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I and II. *Information and Computation*, 100:1–72, 1992.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised 1997*. The MIT Press, 1997.

[NN94] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1994.

[NN95] H. R. Nielson and F. Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In *Proceedings of the Theory and Practice of Software Development (TAPSOFT)*, volume 915 of *LNCS*, 1995.

[NN96a] F. Nielson and H. R. Nielson. Operational semantics of termination types. *Nordic Journal of Computing*, 3:144–187, 1996.

[NN96b] H. R. Nielson and F. Nielson. From CML to process algebras. *Theoretical Computer Science*, 155, 1996.

[NN97] F. Nielson and H.R. Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 332–345. ACM Press, 1997.

[NN98] F. Nielson and H. R. Nielson. Flow logic and operational semantics. In *Electronic Notes in Computer Science*, volume 10. Elsevier Science Publishers, 1998.

[NNH99a] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*, chapter Chapter 3. Springer, 1999.

[NNH99b] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*, chapter 5. Springer-Verlag, 1999.

[PC00] F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 46–57, Montreal,Canada, 2000.

[Plo91] G. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI,Aarhus University, Denmark, 1981(reprinted 1991).

[PO95] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 367–378. ACM Press, 1995.

[PØ97] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, 1997.

[PS95] J. Palsberg and M. I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118:128–141, 1995.

[Rep92] J. Reppy. *Higher-order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.

[Rep99] J. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[RG94] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *LISP and Functional Programming*, pages 65–78, 1994.

[RS01] P. Y. A. Ryan and S. A. Schneider. Process algebra and non-interference. *Journal of Computer Security*, 9(1/2):75–103, 2001.

[San92] D. Sangirogi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[Sew98] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1443 of *LNCS*, pages 695–706. Springer-Verlag, 1998.

[SG90] J. Stamos and D. Gifford. Remote evaluation. *ACM Transactions on Programming Language and Systems*, 12(1):537–565, 1990.

[Sha97] Zhong Shao. An overview of the FLINT/ML compiler. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC)*, 1997.

[Shi91] O. Shivers. *Control-Flow Analysis of Higher-order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie-Mellon University, 1991.

[SS00] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[SSW01] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS)*, LNCS, Sendai,Japan, 2001. Springer-Verlag. To appear.

[Ste96] P. Steckler. Detecting local channels in Distributed Poly/ML. Technical Report ECS-LFCS-96-340, LFCS, University of Edinburgh, 1996.

[SV97] G. Smith and D. Volpano. A type-based approach to program security. In *Proceedings of the Theory and Practice of Software Development (TAP-SOFT)*, pages 607–621, Lille, France, 1997.

[SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, CA, USA, 1998.

[SY97] T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Oriented Distributed Systems (FMOODS)*, volume 2. Chapman and Hall, 1997.

[Tho89] B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Imperial College, London University, 1989.

[Tho94] B. Thomsen. Polymorphic sorts and types for concurrent functional programs. In J. Glauert, editor, *Proceedings of the 6th International Workshop on the Implementation of Functional Languages*, Norwich,UK, 1994.

[TJ92] Y. Tang and P. Jouvelot. Control-flow effects for escape analysis. In *Proceedings of Workshop on Static Analysis (WSA)*, pages 313–321, Bordeux, France, 1992.

[TJ94] J.P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, 1994.

[TLP+93] B. Thomsen, L. Leth, S. Prasad, T. M. Kuo, A. Kramer, and F.C. Knabe. Facile Antigua Release programming guide. Technical Report ECRC-93-20, Europian Computer Industry Research Centre, Munich, Germany, Dec. 1993.

[TMC+96] D. Tarditi, G. Morriset, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimising compiler for ML. In *ACM SIGPLAN '96 Conference on Programming Languages*, pages 181–192, Philadelphia, 1996.

[TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[VS98] D. Volpano and G. Smith. Confinement properties for programming languages. *SIGACT News*, 29(3):33–42, 1998.

[YH99] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, pages 557–572, Santa Barbara, US, 1999. Springer-Verlag.