Type Systems for Access Control and Information Flow in Programming Languages

Marija Kolundžija

Università di Torino
Université de Nice Sophia Antipolis

Thesis directors:
Mariangiola Dezani
Gérard Boudol

Thesis submitted for the Degree of Doctor of Philosophy
Università di Torino & Université de Nice Sophia Antipolis

2008
И уповая буду на Него, и спасусь Им...  

С нами Богъ!
Abstract

This thesis explores language–based security for access control, declassification and information flow in the context of two different languages.

In the first part, we integrate programming constructs for managing confidentiality in an ML-like imperative and higher-order programming language, dealing with both access control and information flow control. Our language includes in particular a construct for declassifying information, and constructs for granting, restricting or testing the read access level of a program. We introduce a type and effect system to statically check access rights and information flow. We show that typable programs are secure, that is, they do not attempt at making illegal read accesses, nor illegal information leakage. This provides us with a natural restriction on declassification, namely that a program may only declassify information that it has the right to read.

In the second part, we integrate security into a service–centered calculus.

The growing importance of service-oriented computing has triggered development of formal computational models for service description and orchestration. Several versions of the Service Centered Calculus (SCC) and its successor, the Calculus of Services with Pipelines and Sessions (CaSPiS) have emerged as outcome of those studies, and are based on the notion of interaction patterns called sessions between the service and the client who invokes it. We propose a security oriented extension of Bruni and Mezzina’s typed variant of CaSPiS, where security levels have been assigned to service definitions, clients and data. In order to invoke a service, a client must be endowed with an appropriate clearance, and once the service and client agree on the security level, the data exchanged in the initiated session will not exceed this level. We study a type system that statically ensures these security properties.
## Contents

**Acknowledgments**

1 Introduction

1.1 Motivation

1.1.1 Integrating Access Control and Information Flow

1.1.2 Security Types for a Service Calculus

2 Background

2.1 Language based security

2.2 Service Calculi

3 Declassification and Access Control

3.1 The language

3.1.1 Security (pre-)lattices

3.1.2 Syntax

3.1.3 Operational Semantics

3.2 The type and effect system

3.3 Secure information flow

3.4 Type System Properties

3.4.1 Soundness
Behavior of “Low”-Terminating Expressions ................................. 20
Behavior of Typable Low Expressions ........................................... 28
Behavior of Typable Expressions ................................................... 33

4 Security Types for Sessions and Pipelines ................................. 34
   4.1 Service Oriented Calculi ..................................................... 34
       4.1.1 Services and Security Issues ......................................... 34
   4.2 The Language ................................................................. 35
       4.2.1 Syntax ................................................................. 35
       4.2.2 Operational Semantics ............................................... 37
   4.3 Type System ................................................................. 39
   4.4 Example ................................................................. 45
   4.5 Properties ................................................................. 46
       4.5.1 Subject Reduction ..................................................... 46
       4.5.2 Security Properties ................................................... 52
   4.6 Progress ................................................................. 54

5 Conclusion ................................................................. 57

Bibliography ................................................................. 60
List of Figures

3.1 Syntax ................................................................. 8
3.2 Reduction ............................................................ 9
3.3 The Type and Effect System ........................................ 13
3.4 The relation $T_{F,low}$ .............................................. 21
3.5 The relation $R_{F,low}$ .............................................. 29

4.1 Syntax ................................................................. 36
4.2 Reduction Rules ..................................................... 37
4.3 Structural Equivalence ............................................ 37
4.4 Syntax of Types ..................................................... 39
4.5 Duality Mapping .................................................... 39
4.6 Subtyping Relation $\leq$ ........................................... 39
4.7 Typing Rules for Inner Processes ................................. 40
4.8 Function pipe ......................................................... 41
4.9 Typing Rules for Top Level Processes ............................ 41
4.10 Typing Rules for Runtime Processes ............................. 41
4.11 Consumption Relation $\ll$ ......................................... 47
4.12 Labeled Operational Semantic Rules .............................. 52
4.13 Security Errors ...................................................... 52
Acknowledgments

I am truly grateful to my PhD supervisor Mariangiola Dezani for the availability, guidance and support she offered me throughout my PhD studies.

Further, I would like to thank the Università Italo-Francese / Université Franco-Italienne for the financial support of my PhD studies through the Vinci integrative program for conjoint PhD curriculums in French-Italian co-tutorship.

I would also like to thank the research group on “Semantics and Logics of Computation” of the Dipartimento di Informatica of the Università di Torino and the MIMOSA group from INRIA, Sophia Antipolis for hosting me during my PhD studies.

Finally, I would like to thank my family, friends and colleagues for their affection, friendship and support which accompanied me in happy as well as hard periods, and made the former more enjoyable, and the latter more endurable.
Chapter 1

Introduction

1.1 Motivation

This work addresses the issues of security and confidentiality in the context of “global” or “ubiquitous” computing. The processors, and consequently the software, are introduced more and more into all aspects of everyday life. This also implies that the communication and the diffusion of information is often carried out by average data processing, toward the subjects a priori unknown. In such a context, where “attacks” of a new type appear (viruses, hacking, etc.), the aspects of safety become crucial. These are the aspects that we address in this thesis. The purpose of this work is to obtain methods of program analysis that aim at guaranteeing properties of security.

1.1.1 Integrating Access Control and Information Flow

A well–known security property is the confidentiality, which ensures that a user (which can be a program) can know only the information which it has the right to access. Mechanisms of access control have for a long time been implemented in the information processing systems. However, this control has been proven to be insufficient [15, 24, 34] because an authorized user can disclose confidential information to other users, thus revealing a secret. This is why the access control must be supplemented by the control of information flow.

So far, the classical way of abstractly specifying secure information flow is to use a lattice of security levels [15]. The “objects” – information containers – of a system are then labeled by security levels, and information is allowed to flow from one object to another if the source object has a lower confidentiality level than the target one. That is, the ordering relation on security levels determines the legal flows, and a program is secure if, roughly speaking, it does not set up illegal flows from inputs to outputs. This was first formally stated via a notion of strong dependency, which is most often referred to as non-interference. There is still a number of issues to investigate in order to make the techniques developed following the language–based approach to information-flow security useful in practice. One of the challenges is declassification. Indeed, there are many useful programs that need to declassify information, from a confidential status to a less secret one. A typical example is a password checking procedure, which delivers to any user the result of comparing a submitted password with secret information contained in a database, thus leaking a bit of confidential information. Such a program is, by definition, ruled out by the non–interference requirement, which is therefore too strong to be used in practice. The problem of taking into account declassifying programs in checking secure information flow has recently motivated a lot of work [1, 3, 30, 31, 36].

In a paper by Almeida Matos and Boudol [3], the authors have introduced a programming construct for managing declassification, that consists in declaring flow policies with a local scope thus defining a new
confidentiality policy, called the *non-disclosure policy*, that generalizes non-interference while allowing one to deal with declassification.

There is no constraint on using the declassification construct in [3], and this is in contrast with most other studies, which aim at restricting the use of such an operation (sometimes without justifying such constraints, for lack of a corresponding extensional notion of security). By putting access control into the picture, we have provided a natural way to restrict declassification, namely: a program may only declassify information that it has the right to read.

In order to do this, we have integrated into the language a formal approach to access control that has recently been introduced to deal with the “stack inspection” mechanism of JAVA security architecture [19, 33, 38]. That is, we add to the language of [3] the constructs for managing access control: restricting, augmenting and testing of the access rights enforced by the context.

Our main results are, first, a type safety property showing that access control is indeed ensured by the type system, that is, a typable program never attempts to read a reference for which it would not be granted the appropriate reading clearance. Second, we extend the soundness result of [3], showing that secure information flow is ensured by our type system. In this way, we achieve the enforcement of “end-to-end confidentiality” in our language, while restricting declassification in a natural way.

This part of the thesis is an expanded version of [10].

### 1.1.2 Security Types for a Service Calculus

Popularity of communication centered applications distributed over the web (*web services*) has provoked urgent interest in the development of automatic tools to assure safe uses of these applications. *Service Oriented Computing* (SOC) has emerged as a new computational paradigm. Services are heterogeneous computational entities, that are developed separately and often are scarcely reliable. Each service has full autonomy in denying a request or abandoning a pending interaction, therefore, a language for SOC should fix some standard mechanism for programming such decisions and to handle their consequences. A language for SOC should support programming of complex and safe client-service interactions. By interaction, we mean the main unit of activity in a service-oriented application, which is essentially a conversation between a client and an instance of a service. The interaction may be complex as it will in general comprise both the exchange of several messages and the invocation of subsidiary services. Orchestration is the process of assembling different services to build a new one or simply to perform a specific computation. A central aspect of orchestration is the organization of the data flow among different activities. This flow also determines synchronization of activities.

A lot of research has been done in this direction, and as a consequence, various process calculi have been designed to model service behaviour (see for example the references in [9]).

An interesting proposal is the *Calculus of Services with Pipelines and Sessions* (CaSPiS) [9], which is a dataflow–oriented successor of the Service Centered Calculus (SCC) [8]. In both these calculi communications can either follow fixed protocols (*session*) between *clients* and *services*, or be disciplined data flows (*pipelines*).

The pipeline constructor of CaSPiS was first introduced in [22]. In [21], the authors have proposed the session as a language construct for communication based programming.

There are also various type disciplines for calculi originated from SCC [25, 26] and in particular for minor modifications of CaSPiS [2, 12]. In both typed versions of CaSPiS the communications between parallel processes are controlled by *session types*, which are lightweight descriptions of protocols first used in [21]. The typed versions of CaSPiS have safe communications, since they enjoy the subject reduction and the progress property (formalized in [18, 17]). Surprisingly, little work has been done to address security issues in the session–type setting. The authors of [7] have proposed correspondence assertions as means of control of data propagation over multiple parties. However, to the best of our knowledge, no notion of secrecy has been considered so far, and therefore no assurance that private data will never become visible to unauthorized bodies can be given.
The second part of this thesis tries to fill this gap by building on well-known typing techniques for controlling access rights [39, 4, 38, 33, 19, 10].

From an access control point of view, clients are regarded as “subjects”, and services and basic values as “objects”. Therefore, a client is required to possess an appropriate clearance to activate a service or to communicate a value. On the other hand, services are regarded as trusted entities, and they are allowed to act on behalf of the client who invoked them (this is reflected in the operational semantics, where the body process of the service definition “inherits” the clearance from the client after the activation).

We have developed a type system, which ensures at compile time that a service can be activated only by clients with appropriate clearance, while preserving the safety and progress properties of previous versions of CaSPiS.

This part of the thesis is an expanded version of [23].
Chapter 2

Background

Early work on the confidentiality problem developed mandatory access control [16]. In this approach, each data item is labeled with a corresponding security level that is a simple confidentiality policy. Information flow is controlled by augmenting the ordinary computation of data within a running program with a simultaneous computation of the corresponding label that controls its future dissemination. This approach, prescribed by the U.S. Department of Defense “orange book” for secure systems, has proved to be too restrictive for general use. It’s additional weakness was its inability to track implicit information flows, the so-called {	extit{covert channels}}.

Further research has come up with static analysis techniques for controlling information flow. These techniques provided a more precise information flow control at compile-time, hence eliminating the run-time overhead.

In the type-checking approach, every program expression has a two-part security type: a basic type (such as boolean or integer, for example), and a static label which defines an information-flow policy on the use of the labeled data. Security is enforced by type checking; the compiler reads a program containing labeled types, and while performing the typechecking procedure, it ensures that the program cannot contain improper information flows at run time.

2.1 Language based security

This thesis focuses on language–based techniques (program semantics and analysis) [41, 34] for the specification and enforcement of security policies for data confidentiality.

The standard security mechanisms are unsatisfactory for protecting confidential information in the emerging, large networked information systems (such as military, medical, and financial information systems, as well as web-based services such as mail, shopping, and business-to-business transactions are applications). The privacy issues which arise in these large-scale systems have not yet been adequately addressed.

There are many aspects to be taken into account when asserting confidentiality properties of a system - from unintentional omissions in the design or implementation to interaction with untrusted (and potentially malicious) entities.

The standard way to protect confidential data used to be (discretionary) access control: some privilege is required in order to access files or objects containing the confidential data. When data is released from its container, the access control mechanism asserts that data has been released to a privileged entity. However, access control mechanism has no power over the propagation of the released data - once it has been taken over by the program, the program could - intentionally or accidentally - transmit it to an untrusted entity. To ensure that information is used only in accordance with the relevant confidentiality policies, it is necessary
to analyze how information is propagated within the using program.

A system is said to be secure with respect to confidentiality as a result of a rigorous analysis that asserts that the system enforces the confidentiality policies of its users. This analysis must show that information controlled by a confidentiality policy cannot flow to a location where that policy is violated. Information-flow policies are a natural way to apply the well-known systems principle of end-to-end design to the specification of computer security requirements; therefore, we also consider them to be specifications of end-to-end security. In a truly secure system, these confidentiality policies could be precisely expressed and translated into mechanisms that enforce them.

An approach to practical methodology for enforcement of information-flow policies that has been developed by the research community is the use of type systems for controlling information flow [11, 13, 30, 27, 31, 34]. In a security-typed language, the types of program variables and expressions are augmented with annotations that specify policies on the use of the typed data.

These security policies are then enforced by compile-time type checking, and, thus, add little or no run-time overhead. Like ordinary type checking, security-type checking is also inherently compositional: secure subsystems combine to form a larger secure system as long as the external type signatures of the subsystems agree. The recent development of semantics based security models (i.e., models that formalize security in terms of program behavior) has provided powerful reasoning techniques about the properties that security-type systems guarantee. These properties increase security assurance because they are expressed in terms of end-to-end program behavior and, thus, provide a suitable vocabulary for end-to-end policies of programs.

2.2 Service Calculi

An important group of calculi for modeling and proving properties of services is the one based on the explicit notion of session [21]. A session corresponds to a private channel that is instantiated when calling a service: it binds caller and callee and is used for their communication.

To manage inter-session communication, different mechanisms have been proposed that are at the moment under evaluation. The first model was a single core calculus (SCC) [8], whose successors are variants that have branched on different inter-session communication mechanisms. SCC has been influenced by Cook and Misra’s Orc [22], a basic programming model for structured orchestration of services, and by \( \pi \)-calculus [29], the by now classical representative of name passing calculi. Indeed one could say that SCC combines the service-oriented flavor of Orc with the name passing communication mechanism of \( \pi \)-calculus. In particular, Orc has been appealing to us because of its simplicity and yet great generality: its three basic composition operators can be used to model the most common workflow patterns; SCC emerged as a result of combining descriptive power similar of Orc but with the mathematical cleanness of \( \pi \)-calculus.

SCC supports explicit modeling of sessions that are rendered as private bi-directional channels created upon services invocation and used to bind caller and callee. The interaction is programmed by two communication protocols installed at each side of the bi-directional channel. This session mechanism permits describing and reasoning about interaction patterns that are more structured than the classical one-way and request-response pattern. Essentially, SCC is a name passing process calculus with explicit notions of service definition, service invocation and bi-directional sessioning.

Within SCC, services are seen as sort of interacting functions (and even stream processing functions) that can be invoked by clients. A service invocation causes activation of a new session. Client and service protocols are then instantiated each at the proper side of the session. More generally, within sessions communication is bi-directional, in the sense that the interacting protocols can exchange data in both directions. Values returned outside the session to the enclosing environment can be used for invoking other services.

The original proposal of SCC was somehow unsatisfactory with respect to the handling of intersession communications, thus variants of SCC that make use of different communication mechanisms have been put forward.
• CaSPiS [9] is dataflow oriented and makes use of a pipelining operator (à la ORC) to model the passage of information between sessions; return is used by sessions for passing values to the environment.

• SCC [25] is stream oriented and has primitives for inserting/retrieving data in/from streams that are used both for inter-session communication and for communication with the environment.

• CSCC [26] is message oriented and has explicit and distinct message passing primitives to model inter and intra session communication and for communicating with the environment.

CaSPiS is variant of SCC that is dataflow oriented and makes use of a pipelining operator to model the passage of information between sessions. Specific care is devoted to formal semantics and minimality of the operators. A session is a chain of dyadic interactions whose collection constitutes a program. Services are seen as passive objects that can be invoked by clients and service definitions can be seen as specific instances of input prefixed processes. The two endpoints of the same session can communicate by exchanging messages. A fresh shared name is used to guarantee that messages are exchanged only between partners of the same session, so that two instances of the same persistent service (that was invoked from two different sessions) run separately and cannot interfere. The central role assigned to sessions and the direct use of operators for modeling sessions interaction renders the logical structure of programs more clear and leads to a well disciplined service specification language that enable us to guarantee proper handling of session closures and in general simplifies reasoning on the specified services.
Chapter 3

Declassification and Access Control

We shall address combining access control and information flow in the presence of declassification. The purpose is to introduce, in an imperative (higher-order) lambda-calculus with thread and reference creation, a flow declaration construct that locally extends the global flow policy, thus providing a mechanism for expressing declassification within the scope of the declaration, constructs for constraining, enabling and testing access rights.

Then we present the non-disclosure policy dependent on the access privilege, and show that introduction of access control adds a way of restricting declassification – a program can declassify only what it has the right to read.

We present a type and effect system for enforcing this policy.

3.1 The language

3.1.1 Security (pre-)lattices

The security levels are hierarchically organized in a pre-lattice, a structure defined as a pair \((L, \preceq)\), where \(\preceq\) is a preorder relation over the set \(L\), that is a reflexive and transitive, but not necessarily symmetric relation, such that for any \(x, y \in L\) there exist a meet \(x \land y\) and a join \(x \lor y\) satisfying:

\[
\begin{align*}
x \land y &\preceq x \\
x \land y &\preceq y \\
z \preceq x \land z \preceq y &\Rightarrow z \preceq x \land y \\
x \preceq z \land y \preceq z &\Rightarrow x \lor y \preceq z
\end{align*}
\]

The pre-lattices we use are defined as follows. We assume given a set \(\text{Pri}\) of principals, ranged over by \(p, q\ldots\) (From an access control perspective, these are also called permissions \([5, 19]\), or privileges \([33, 38]\), while a “principal” is a set of permissions.) A confidentiality level is any set of principals, that is any subset \(\ell\) of \(\text{Pri}\). The intuition is that whenever \(\ell\) is the confidentiality label of an object, i.e. a reference, it represents a set of programs that are allowed to get the value of the object, i.e. to read the reference. Then a confidentiality level is similar to an access-control list (i.e. a set of permissions). From this point of view, a reference labelled \(\text{Pri}\) (also denoted \(\bot\)) is the most public one – every program is allowed to read it – whereas the label \(\emptyset\) (also denoted \(\top\)) indicates a secret reference, and reverse inclusion of security levels may be interpreted as indicating allowed flows of information: if a reference \(u\) is labelled \(\ell\), and \(\ell \supseteq \ell'\), then the value of \(u\) may be transferred to a reference \(v\) labelled \(\ell'\), since the programs allowed to read this value from \(v\) were already allowed to read it from \(u\).

We follow the approach of \([3]\), where declassification is achieved by dynamically updating the lattice.
structure of confidentiality levels, by the means of local flow policies. A flow policy is a binary relation over Pri. We let F, G . . . range over such relations. A pair (p, q) ∈ F is to be understood as “information may flow from principal p to principal q”, that is, more precisely, “everything that principal p is allowed to read may also be read by principal q”. As a member of a flow policy, a pair (p, q) will be written p ≺ q. We denote, as usual, by F∗ the preorder generated by F (that is, the reflexive and transitive closure of F). Any flow policy F determines a preorder on confidentiality levels that extends reverse inclusion, as follows:

\[ \ell \preceq_F \ell' \iff \forall q \in \ell'. \exists p \in \ell. p F^* q \]

It is not difficult to see that the preorder \( \preceq_F \) induces a pre-lattice structure on the set of confidentiality levels, where a meet is simply the union, and a join of \( \ell \) and \( \ell' \) is

\[ \{ q \mid \exists p \in \ell. \exists p' \in \ell'. p F^* q \& p' F^* q \}. \]

This observation justifies the following definition.

**Definition (Security Pre-Lattices)** 1.1. A confidentiality level is any subset \( \ell \) of the set Pri of principals. Given a flow policy \( F \subseteq Pri \times Pri \), the confidentiality levels are pre-ordered by the relation

\[ \ell \preceq_F \ell' \iff \forall q \in \ell'. \exists p \in \ell. p F^* q \]

The meet and join, w.r.t. \( F \), of two security levels \( \ell \) and \( \ell' \) are respectively given by \( \ell \cup \ell' \) and

\[ \ell \cap_F \ell' = \{ q \mid \exists p \in \ell. \exists p' \in \ell'. p F^* q \& p' F^* q \} \].

### 3.1 Syntax

The syntax of the language considered in this chapter is given in Figure 3.1. The language is a higher-order \( \lambda \)-calculus, extended with imperative constructs of ML, conditional branching and boolean values. The language is further enriched with constructs for dynamic manipulation and testing of access rights ([5, 19, 33, 38]), and dynamic extension of the flow policy ([3]). The language consists of expressions and values, where values are the expressions that cannot be further computed. The values include variables, functions, boolean values `true` and `false`, memory references and a nil command `()`, used to denote inactivity.

The expressions include:

- **conditional branching** if \( M \) then \( N \) else \( N' \) on a boolean value \( M \), which executes \( N \) or \( N' \), depending on evaluation of \( M \) being `true` or `false`;
- **application** \( (MN) \) of function obtained by evaluation of \( M \) to the result of evaluation of \( N \);
- **the sequential composition** \( M ; N \), which executes \( N \) after the termination of \( M \);
- **reference creation** \( (ref_{\ell,\theta} N) \), which creates a new memory location of type \( \theta \) and with a security level \( \ell \) in which the value obtain by evaluation of \( N \) is written;
- **dereferencing** \( (! N) \), which reads the reference obtained by evaluation of \( N \);

\[
M, N \ldots \in \text{Exp} \quad ::= \quad V \mid \text{if } M \text{ then } N \text{ else } N' \mid (MN) \quad \text{expressions}
\]

\[
\mid M ; N \mid (\text{ref}_{\ell,\theta} N) \mid (! N) \mid (M := N)
\]

\[
\mid (\ell \bowtie M) \mid (\text{enable } \ell \text{ in } M) \mid (\text{test } \ell \text{ then } M \text{ else } N)
\]

\[
\mid (\text{flow } F \text{ in } M)
\]

\[
V \in \text{Val} \quad ::= \quad x \mid u_{\ell,\theta} \mid gx.M \mid \lambda x.M \mid \text{true} \mid \text{false} \mid () \quad \text{values}
\]

Figure 3.1: Syntax
The reduction relation is a transition relation between configurations of the form $M, \mu$.

### 3.1.3 Operational Semantics

As usual, denote by $\text{dom}(\mu)$ the name $u$ is fresh for $\nu$, whereby $\nu$, ranged by $\mu$, is defined as a mapping mapping from a finite set $\text{dom}(\mu)$ of references to values.

The reduction relation is a transition relation between configurations of the form $(M, \mu)$ where $M$ is an expression, and $\mu$ is the memory.

The operation of updating the value of a reference in the memory is denoted as $\mu[\ell, \theta := V]$. We say that the name $u$ is fresh for $\mu$ if $\ell, \theta \in \text{dom}(\mu) \Rightarrow v \neq u$.

In what follows we shall only consider well-formed configurations, that is pairs $(M, \mu)$ such that $\text{loc}(M) \subseteq \text{dom}(\mu)$ and for any $\ell, \theta \in \text{dom}(\mu)$ we have $\text{loc}(\mu[\ell, \theta]) \subseteq \text{dom}(\mu)$ (this property will be preserved by the

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IFT) $\ell \vdash_G (\text{E}[\text{if true then } M \text{ else } N], \mu) \rightarrow (\text{E}[M], \mu)$</td>
<td></td>
</tr>
<tr>
<td>(IFF) $\ell \vdash_G (\text{E}[\text{if false then } M \text{ else } N], \mu) \rightarrow (\text{E}[N], \mu)$</td>
<td></td>
</tr>
<tr>
<td>(APP) $\ell \vdash_G (\text{E}[(\lambda x.MV)], \mu) \rightarrow (\text{E}[x \mapsto V], \mu)$</td>
<td></td>
</tr>
<tr>
<td>(REC) $\ell \vdash_G (\text{E}[(\text{gx}.M)], \mu) \rightarrow (\text{E}[x \mapsto \text{gx}.M], \mu)$</td>
<td></td>
</tr>
<tr>
<td>(SEQ) $\ell \vdash_G (\text{E}[V; N], \mu) \rightarrow (\text{E}[N], \mu)$</td>
<td></td>
</tr>
<tr>
<td>(REF) $\ell \vdash_G (\text{E}[\text{ref}\ell, \theta V], \mu) \rightarrow (\text{E}[\text{ref}\ell, \theta {\mu[\ell, \theta] := V}], \mu)$ if $\ell$ fresh for $\mu$</td>
<td></td>
</tr>
<tr>
<td>(DEREF) $\ell \vdash_G (\text{E}[\ell \text{ uv}, \theta V], \mu) \rightarrow (\text{E}[V], \mu)$</td>
<td></td>
</tr>
<tr>
<td>(ASSIGN) $\ell \vdash_G (\text{E}[\ell \text{ uv}, \theta := V], \mu) \rightarrow (\text{E}[\ell \text{ uv}, \theta := V], \mu)$</td>
<td></td>
</tr>
<tr>
<td>(SIGN) $\ell \vdash_G (\text{E}[\ell \times V], \mu) \rightarrow (\text{E}[\ell \times V], \mu)$</td>
<td></td>
</tr>
<tr>
<td>(ENABLE) $\ell \vdash_G (\text{E}[\text{enable } \ell \text{ in } V], \mu) \rightarrow (\text{E}[V], \mu)$</td>
<td></td>
</tr>
<tr>
<td>(TESTT) $\ell \vdash_G (\text{E}[\text{test } \ell \text{ then } M \text{ else } N], \mu) \rightarrow (\text{E}[M], \mu)$ if $\ell \leq_G [\text{E}]$</td>
<td></td>
</tr>
<tr>
<td>(TESTF) $\ell \vdash_G (\text{E}[\text{test } \ell \text{ then } M \text{ else } N], \mu) \rightarrow (\text{E}[N], \mu)$ if $\ell \not\leq_G [\text{E}]$</td>
<td></td>
</tr>
<tr>
<td>(FLOW) $\ell \vdash_G (\text{E}[\text{flow } F \text{ in } V], \mu) \rightarrow (\text{E}[V], \mu)$</td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 3.2: Reduction

- **Reference updating** $(M := N)$, which updates the reference obtained by evaluation of $M$ with the result of evaluation of $N$;
- **Access restriction** $(\ell \times M)$ (inspired by the “framed” expressions of [19], and by the “signed” expressions of [33]), used to restrict the access right of $M$ to $\ell$; this is a scoping construct: the current reading clearance is restored after termination of $M$;
- **Access extension** (enable $\ell$ in $M$), used to locally extend the read access right of $M$ by $\ell$; it is also a scoping construct;
- **Local flow declaration** (flow $F$ in $M$), which enables extension of the current flow policy by a local policy $F$ while reducing $M$, usually for declassification purposes.

A reference is a location name $u$ to which a confidentiality level $\ell$ is assigned. For technical reasons, we also record the type $\theta$ (see Section 3 below) of the contents of the reference.

We denote by $\text{loc}(M)$ the set of decorated locations $u_{\ell, \theta}$ occurring in $M$. These references are regarded as providing the inputs of the expression $M$. We let $\text{fv}(M)$ be the set of variables occurring free in $M$, and we denote by $[\ell\theta\mathcal{V}]M$ the capture-avoiding substitution of $V$ for the free occurrences of $x$ in $M$, where $V \in \mathcal{V}$. As usual, (let $x = N$ in $M$) denotes $(\lambda x.MN)$.

#### 3.1.3 Operational Semantics

Memory, or heap, ranged by $\mu$, $\nu$, is defined as a mapping mapping from a finite set $\text{dom}(\mu)$ of references to values.

The reduction relation is a transition relation between configurations of the form $(M, \mu)$ where $M$ is an expression, and $\mu$ is the memory.

The operation of updating the value of a reference in the memory is denoted as $\mu[u_{\ell, \theta} := V]$. We say that the name $u$ is fresh for $\mu$ if $v_{\ell, \theta} \in \text{dom}(\mu) \Rightarrow v \neq u$.

In what follows we shall only consider well-formed configurations, that is pairs $(M, \mu)$ such that $\text{loc}(M) \subseteq \text{dom}(\mu)$ and for any $u_{\ell, \theta} \in \text{dom}(\mu)$ we have $\text{loc}(\mu[u_{\ell, \theta}]) \subseteq \text{dom}(\mu)$ (this property will be preserved by the
The operational semantics consists in reducing a redex (reducible expression) inside an evaluation context (see [40]). Reducible expressions are given by the following grammar:

\[ R ::= \text{if true then } M \text{ else } N \mid \text{if false then } M \text{ else } N \mid (\mu x.M) \mid (\lambda x.MV) \]
\[ \mid V; N \mid (\text{ref}_{\ell,\theta} V) \mid (\ell u_{\ell,\theta}) \mid (u_{\ell,\theta} := V) \]
\[ \mid (\ell \times V) \mid (\text{enable } \ell \text{ in } V) \mid (\text{test } \ell \text{ then } M \text{ else } N) \]
\[ \mid (\text{flow } \ell \text{ in } V) \]

Evaluation contexts are defined as expression shapes that contain a special expression called a hole, that determines the precedence of reduction. The grammar for evaluation contexts is as follows:

\[ E ::= [] \mid E[F] \]
\[ F ::= \text{if } [] \text{ then } M \text{ else } N \mid ([] N) \mid (V[]) \]
\[ \mid [] ; N \mid (\text{ref}_{\ell,\theta}[]) \mid (\ell []) \mid ([]} := N) \mid (V := []) \]
\[ \mid (\ell \times []) \mid (\text{enable } \ell \text{ in } []) \mid (\text{flow } F \text{ in } []) \]

The operational semantics of an expression depends upon a given global flow policy \( G \) and a default confidentiality level \( \ell \), which represents the current access level, we define \( \ell \vdash G (M, \mu) \to (M', \mu') \)

Given a global flow policy \( G \), for any confidentiality level \( \ell \) representing the current access level, we define the level granted by the evaluation context \( E \), denoted \([E]_{\ell}\), as follows:

\[ [E]_{\ell} \]
\[ [E[F]]_{\ell} = \begin{cases} [E]_{\ell} & \text{if } F = (\text{flow } F \text{ in } []) \\ (E)_{\ell} \lambda_G \ell' & \text{if } F = (\text{test } \ell \text{ in } []) \\ (E)_{\ell} \gamma_G \ell' & \text{if } F = (\text{enable } \ell \text{ in } []) \\ [E]_{\ell} & \text{otherwise} \end{cases} \]

Computing \([E]_{\ell}\) is a form of “stack inspection,” see [19, 33, 38].

We will also be using the flow policy \([E]\) granted by the evaluation context \( E \), which is defined as follows:

\[ [E] = \emptyset \]
\[ [E[F]] = \begin{cases} [E] \cup F & \text{if } F = (\text{flow } F \text{ in } []) \\ [E] & \text{otherwise} \end{cases} \]

The reduction rules are given in Figure 3.2. They are fairly standard, as regards the functional and imperative fragment of the language. One may observe that (flow \( F \) in \( M \)) behaves exactly as \( M \), and that the semantics of the constructs for managing access rights are as one might expect, given the definition of \([E]_{\ell}\) above. We denote by \( \xrightarrow{\ell} \) the reflexive and transitive closure of the reduction relation. More precisely, we define:

\[ \ell \vdash G (M, \mu) \xrightarrow{\ell} (M'', \mu'') \]
\[ \ell \vdash G (M, \mu) \xrightarrow{\ell} (M', \mu') \]

In order to prove certain properties of the type system presented in Section 3.2, we will need to refer to the “current” flow policy or the “current” access right, i.e. the local flow policy or access right that holds for the current computation step, and is imposed by the evaluation context. To facilitate this reference, we introduce the decorated reduction relation, of the form \( \ell \vdash G (M, \mu) \xrightarrow{\ell} (M', \mu') \) and \( \ell \vdash G (M, \mu) \xrightarrow{\ell} (M', \mu') \),
where $F$ is the flow policy and $j$ is the access right that holds for the current this step. More precisely we define:

$$\ell \vdash_G (R, \mu) \rightarrow (M, \mu')$$

$$\ell \vdash_G (E[R], \mu) \xrightarrow{[E]} (E[M], \mu')$$

$$\ell \vdash_G (R, \mu) \rightarrow (M, \mu')$$

$$\ell \vdash_G (E[R], \mu) \xrightarrow{[E]} (E[M], \mu')$$

provided that $R$ is any redex.

An expression is said to converge if, regardless of the memory, its evaluation terminates on a value, that is:

$$M \Downarrow \iff \forall \mu \exists V \in \text{Val} \exists \mu'. (M, \mu) \rightarrow^* (V, \mu')$$

One can see that a constraint that could block the reduction is the dynamic checking of read access rights, that is the condition $\ell' \preceq_G [E] \ell$ when reading a reference $u \ell, \theta$ in the memory, with $\ell$ as the given reading clearance. (Since we are only considering well-formed configurations, the value $\mu(u \ell, \theta)$ is always defined.) Indeed for instance the expression (omitting the types attached to references)

$$(\text{flow } H < L \text{ in } u_L := ! u_H)$$

is blocked if the current access right is $\{L\}$, and can only proceed if this right is at least $\{H\}$. This indicates that, using (flow $F$ in $M$), one can only declassify information that one has the right to read. This is because the local flow policy declarations do not interfere with access control, i.e. they do not play any role in the definition of $[E] \ell$. More generally, to let information flow, such as in $v \ell := ! u \ell$, a program must have the right to read it.

One can easily check the usual property (see [40]) that, given a pair $(\ell, G)$, a closed expression is either a value, or a reducible expression, or is a faulty expression, either for typing reasons, or because it does not have the appropriate access rights, that is:

**Lemma 3.1.1** Let $M$ be a closed expression. Then, for any $\ell$ and $G$, either $M \in \text{Val}$, or for any $\mu$ there exist $F$, $M'$ and $\mu'$ such that $\ell \vdash_G (M, \mu) \rightarrow (M', \mu')$, or $M$ is $(\ell, G)$-faulty, that is:

(i) $M = E[\text{if } V \text{ then } N \text{ else } N']$ and $V \notin \{\text{true, false}\}$, or

(ii) $M = E[(V V')]$ and $V$ is not a functional value $\lambda x. M'$, or

(iii) $M = E[(! V)]$ or $M = E[(V := V')]$ where $V$ is not a reference $u \ell, \theta$, or

(iv) $M = E[(! u \ell, \theta)]$ with $j \not\in_G [E] \ell$.

A key property of our calculus is that by reducing the same expression is two different memories, if we obtain different expressions but the new memory locations created by the reduction coincide, then the redex was a dereferencing and the memories are unchanged.

**Lemma 3.1.2** (Splitting Computations) If $\ell \vdash_G (M, \mu_1) \rightarrow (M_1, \mu'_1)$ and $\ell \vdash_G (M, \mu_2) \rightarrow (M_2, \mu'_2)$, with $M_1 \neq M_2$, and $\text{dom}(\mu'_1) = \text{dom}(\mu_1) = \text{dom}(\mu'_2) = \text{dom}(\mu_2)$, then there exist $E$ and $u \ell, \theta$, such that $M = E[! u \ell, \theta]$, and $M_1 = E[\mu_1(u \ell, \theta)]$, $M_2 = E[\mu_2(u \ell, \theta)]$, with $\mu_1 = \mu'_1$ and $\mu_2 = \mu'_2$.

The condition $\text{dom}(\mu'_1) = \text{dom}(\mu_1) = \text{dom}(\mu'_2) = \text{dom}(\mu_2)$ is necessary for the validity of previous lemma, take for example $\ell \vdash_G (\text{ref } u \ell, \theta V, \mu) \rightarrow (u \ell, \theta, \mu \cup [u \ell, \theta V])$ and $\ell \vdash_G (\text{ref } u \ell, \theta V, \mu) \rightarrow (u \ell, \theta, \mu \cup [u \ell, \theta V])$.

We conclude this section by two examples that illustrate the (decorated) reduction relation.
Example 3.1.1 The first example shows the local behaviour of the flow construct. We consider the reduction of the process:

\[(\text{test } p) \text{ then } (\text{flow } p \prec q \text{ in } v(q), \theta := ! u_{(p, q)} \text{ else } )\]

assuming that \(p \in \ell, \mu(u_{(p)}) = V, \ V\) being of type \(\theta\), and omitting \(\theta\) from the reference decoration.

1. \(\ell \vdash_G (\text{test } p)\) then \((\text{flow } p \prec q \text{ in } v(q) := ! u_{(p)}, \mu) \rightarrow (\text{TestT})\)
2. \(\ell \vdash_G (\text{flow } p \prec q \text{ in } v(q) := ! u_{(p)}, \mu) \rightarrow (\text{Deref})\)
3. \(\ell \vdash_G (\text{flow } p \prec q \text{ in } v(q) := V), \mu) \rightarrow (\text{Assign})\)
4. \(\ell \vdash_G ((\text{flow } p \prec q \text{ in } \mu(v(q) := V)) \rightarrow (\text{Flow})\)
5. \(\ell \vdash_G (\mu, v(q) := V)\)

Example 3.1.2 This is an example of an expression with dynamic use of the test construct. We consider the reduction of the process:

\[\text{let } x = \lambda y (\text{test } x \text{ then } v_1 := ! u_1 \text{ else } )\text{ in}
\text{if } ! u_1' \\text{ then } (\text{enable } x \text{ in } x()) \\text{ else } x()\]

assuming that \(\mu(u_1) = V, \ V\) and all of the references are of type \(\theta\), \(\ell \leq \iota\) and \(\mu(u_1') = \text{true}\).

\[\ell \vdash_G (\text{let } x = \lambda y (\text{test } x \text{ then } v_1 := ! u_1 \text{ else } )\text{ in}
\text{if } ! u_1' \\text{ then } (\text{enable } x \text{ in } x()) \\text{ else } x()\]

\[\ell \vdash_G (\text{let } x = \lambda y (\text{test } x \text{ then } v_1 := ! u_1 \text{ else } )\text{ in}
\text{if } ! u_1' \\text{ then } (\text{enable } x \text{ in } x()) \\text{ else } x()\]

\[\ell \vdash_G (\text{let } x = \lambda y (\text{test } x \text{ then } v_1 := ! u_1 \text{ else } )\text{ in}
\text{if } ! u_1' \\text{ then } (\text{enable } x \text{ in } x()) \\text{ else } x()\]

3.2 The type and effect system

Our main aim in this chapter is to show that we can design a type system that guarantees both secure information flow, as in [3], and, as in [5, 33], the fact that a well-typed expression is never stuck, and therefore that the run-time checking of the access rights is useless for such expressions.

Our type system elaborates on the one of [3], and, as such, is actually a type and effect system [28]. This is consistent with our “state-oriented” approach – as opposed to the “value-oriented” approach of [19, 32, 33, 38] for instance – where only the access to the “information containers”, that is, to the references in the memory, is protected by access rights. In particular, a value is by itself neither “secret” nor “public,” and the types do not need to be multiplied by the set of confidentiality levels. Then the types are

\[\tau, \sigma, \theta, \ldots := \text{bool} \mid \text{unit} \mid \theta \text{ref}_\ell \mid (\tau \xrightarrow{\ell, F} \sigma)\]

where \(s\) is any “security effect” – see below. Notice that a reference type \(\theta \text{ref}_\ell\) records the type \(\theta\) of values the reference contains, as well as the “region” \(\ell\) where it is created, which is the confidentiality level at which the reference is classified. Since a functional value wraps a possibly effectful computation, its type records this latent effect [28], which is the effect the function may have when applied to an argument. It also records the “latent reading clearance” \(\ell\) and the “latent flow policy” \(F\), which are assumed to hold when the function is called.

The judgements of the type and effect system have the form

\[\ell; F; \Gamma \vdash_G M : s, \tau\]

where \(\Gamma\) is a typing context, assigning types to variables, and \(s\) is a security effect, that is a triple \((\ell_0, \ell_1, \ell_2)\).
of confidentiality levels. The intuition is:

- $\ell$ is the current read access right that is in force when reducing $M$;
- $F$ is the current flow policy, while $G$ is the given global flow policy;
- $\ell_0$, also denoted by $s,c$, is the confidentiality level of $M$. This is an upper bound (up to the current flow relation) of the confidentiality levels of the references the expression $M$ reads that may influence

```

\begin{center}

\begin{tabular}{c}
\hline
$\ell; F; \Gamma \vdash_G \text{ref}_{\ell,\theta} : \bot, \theta \text{ref}_F$ & (Loc) & $\ell; F; \Gamma, x : \tau \vdash_G x : \bot, \tau$ & (VAR) \\
\hline
$\ell; F; \Gamma, x : \tau \vdash_G M : s, \sigma$ & (ABS) & $\ell; F; \Gamma \vdash_G \emptyset : \bot, \text{unit}$ & (NIL) \\
\hline
$\ell'; F'; \Gamma \vdash_G \lambda x. M : \bot, (\tau \to s)_{\ell,F}$ & (BOOLT) & $\ell; F; \Gamma \vdash_G \text{tt} : \bot, \text{bool}$ & (BOOLF) \\
\hline
$\ell; F; \Gamma \vdash_G M : s, \text{bool}$ & (COND) & $\ell; F; \Gamma \vdash_G N : s, \tau \quad s.t \leq_{F \cup G} G s, w \cup s, w$ & (SEQ) \\
\hline
$\ell; F; \Gamma \vdash_G \text{if } M \text{ then } N_0 \text{ else } N_1 : s, \tau \quad s.t \leq_{F \cup G} G s, \emptyset \cup s, \emptyset$ & (APP) & $\ell; F; \Gamma \vdash_G N : s, \tau$ & (DEREF) \\
\hline
$\ell; F; \Gamma \vdash_G \text{let } x : \tau \text{ in } M : s, \tau$ & (ASSIGN) & $\ell; F; \Gamma \vdash_G (M := N) : (\bot, s.w \cup s.w \cup \ell', s.t \leq s'.t), \text{unit}$ & (ENABLE) \\
\hline
$\ell \land G \ell'; F; \Gamma \vdash_G M : s, \tau$ & (RESTR) & $\ell; G \ell'; F; \Gamma \vdash_G M : s, \tau$ & (REC) \\
\hline
$\ell; F; \Gamma \vdash_G (\ell' \times M) : s, \tau$ & (TEST) & $\ell; F; \Gamma \vdash_G \text{if } \ell' \text{ then } M \text{ else } N : s, \tau$ & (FLOW) \\
\hline
$\ell; F; \Gamma \vdash_G \text{test } \ell' \text{ then } M \text{ else } N : s, \tau$ & (RESTR) & $\ell; F; \Gamma \vdash_G \text{let } x : \tau \text{ in } W : s, \tau$ & (REC) \\
\hline
$\ell; F; \Gamma \vdash_G \text{let } x : \tau \text{ in } W : s, \tau$ & (TEST) & $\ell; F; \Gamma \vdash_G (\ell' \times M) : s, \tau$ & (FLOW) \\
\hline
\end{tabular}

\end{center}

Figure 3.3: The Type and Effect System
```
its resulting value;

- $\ell_1$, also denoted $s.w$, is the writing effect, that is a lower bound (w.r.t. the relation $\preceq$) of the level of references that the expression $M$ may update;

- $\ell_2$, also denoted $s.t$, is an upper bound (w.r.t. the current flow relation) of the levels of the references the expression $M$ reads that may influence its termination. We call this the termination effect of the expression.

In the following we shall denote $s.c \vdash F \cup G . s.t$ by $s.r$, assuming that $F$ and $G$ are understood from the context. According to the intuition above, the security effects $s = (c, w, t)$ are ordered componentwise, in a covariant manner as regards the confidentiality level $c$ and the termination effect $t$, and in a contravariant way as regard the writing effect $w$. Then we abusively denote by $\bot$ and $\top$ the triples $(\bot, \bot, \bot)$ and $(\top, \bot, \top)$ respectively. In the typing rules for compound expressions, we will use the join operation on security effects:

$$s \cdot F \cdot s' =_{def} (s.c \cdot F \cdot s'.c, s.w \cup s'.w, s.t \cdot F \cdot s'.t)$$

as well as the following convention:

**Convention.** In the type system, when the security effects occurring in the context of a judgement $\ell; F; \Gamma \vdash G : s, t$ involve the join operation $\cdot$, it is assumed that the join is taken w.r.t. $F \cup G$, i.e. it is $\cdot F \cup G$. We recall that by $s.r$ we mean $s.c \cdot F \cup G . s.t$.

The typing system is given in Figure 3.3. Notice that this system is syntax-directed: there is exactly one rule per construction of the language. In particular, there is no subtyping rule. As an example, we infer the types for the expressions of Example 3.1.1 and Example 3.1.2.

**Example 3.2.1** For the expression:

$$\text{if } F = G \cup \{ p \prec q \}, \text{ the typing is as follows:}$$

$$\{ p \}; F; \Gamma \vdash G : (\bot, \bot, \bot)$$

if $p = G \cup \{ p \prec q \}$, the typing is as follows:

$$\{ p \}; F; \Gamma \vdash G : (\bot, \bot, \bot)$$

**Example 3.2.2** For space reasons we split the typing of the expression:

$$\text{let } x = \lambda y (\text{test } \ell \text{ then } v_\ell \text{ else } \bot) \text{ in}$$

if $M$ then (enable $\ell$ in $x()$) else $x()$
Finally, we type the application:

\[ \ell; G; \Gamma \vdash \lambda x \text{ if } ! u'_{\ell} \text{ then } (\text{enable } s \text{ in } x()) \text{ else } x() : \bot, \tau \]  
\[ \ell; G; \Gamma \vdash \lambda y (\text{test } s \text{ then } \nu_{\ell} := ! u_{\ell} \text{ else }()) : \bot, \tau \]  
\[ \ell; G; \Gamma \vdash (\lambda x \text{ if } ! u'_{\ell} \text{ then } (\text{enable } s \text{ in } x()) \text{ else } x())(\lambda y (\text{test } s \text{ then } \nu_{\ell} := ! u_{\ell} \text{ else }())) : (\ell', \bot, \bot) \]  

\[ \text{APP} \]

Compared to the system of [3], which does not involve constructs for managing access control, the main difference is in the (DEREF) rule, where we have the constraint that the level of the reference that is read should be less than the access level granted by the context. Notice that this constraint only involves the global flow policy \( G \), not the local one \( F \). This is the way to ensure that declassification does not modify the access rights. There is a similar constraint in the rule for typing application, where the access level required by the (body of the) function should indeed be granted by the context. It is easy to see that typing enjoys a “weakening” property, asserting that if an expression is typable in the context of some access right \( \ell \), then it is also typable in the context of a more permissive reading clearance. Similarly, one could show that relaxing (that is, extending) the global or local flow policy does not affect typability.

**Lemma 3.2.1**

- If \( \ell; F; \Gamma \vdash M : s, \tau \) and \( \ell \leq G, \ell' \), then \( \ell' ; F; \Gamma \vdash M : s, \tau \).
- If \( \ell; F; \Gamma \vdash M : s, \tau \), then \( \ell; F; \Gamma \vdash G, \ell' : M : s, \tau \).
- If \( \ell; F; \Gamma \vdash M : s, \tau \), then \( \ell; F \cup F'; \Gamma \vdash M : s, \tau \).

**Proof:** By induction on the inference of the typing judgement.

### 3.3 Secure information flow

In [3], the authors have proposed a generalization of the usual non-interference property that allows one to deal with declassification. The idea is to define a program as secure if it satisfies a “local non-interference” property, called the *non-disclosure policy*, which roughly states that the program is, at each step of its execution, non-interfering with respect to the current flow policy, that is the global flow policy extended by the one granted by the evaluation context. Technically, a program will be considered as secure if it is isomorphous to itself. As usual, this property relies on preserving the “low equality” of memory. Roughly speaking, two memories are equal up to level \( \ell \) if they assign the same value to every location with security level lower than \( \ell \), with respect to a given flow policy. In order to deal with reference creation, we only compare memories on the domain of references they share – then the “low equality” of memory is actually not an equivalence (it is not transitive). Nevertheless, we keep the standard terminology. The “low equality” of memories, with respect to a current flow policy \( F \), and to a security level \( \ell \) regarded as “low,” is thus defined:

\[ \mu \triangleright^{F, \ell} \nu \iff_{\text{def}} \forall u'_{\ell, \gamma} \in \text{dom}(\mu) \cap \text{dom}(\nu), \ell' \leq \ell \Rightarrow \mu(u'_{\ell, \gamma}) = \nu(u'_{\ell, \gamma}) \]

From an information flow point of view, the notion of a secure program actually depends on the current access right. For instance, the assignment \( \nu_{\ell} := ! u_{\ell} \) where \( \ell \not\leq \ell' \), which is usually taken as a typical
example of an unsecure program, is indeed secure (in the sense of [14]) in the context of a current access level \( \ell'' \) such that \( \ell \not\preceq_G \ell'' \) (but in that case this program attempts a confidentiality violation, as regards access control).

Our definition of secure programs is parameterized by a global flow policy.

**Definition 3.3.1 (Bisimulation)** A \((G, \ell)\)-bisimulation is a symmetric relation \( R \) on expressions such that if \( M R N \) and \( \kappa \vdash_G (M, \mu) \rightarrow (M', \mu') \) with \( M = E[R] \) where \( R \) is a redex, and if \( \nu \) is such that \( \nu \cup G \cup \{E\} \) and \( u_{\ell', \theta} \in \text{dom}(\mu') - \text{dom}(\mu) \) implies that \( u \) is fresh for \( \nu \), then there exist \( N' \) and \( \nu' \) such that \( \kappa \vdash_G (N, \nu) \rightarrow (N', \nu') \) with \( M' R N' \) and \( \mu' \equiv_G \nu' \).

**Remark 3.3.2 (Remarks and Notation)**
(i) For any \( \ell, G \) there exists a \((G, \ell)\)-bisimulation, like for instance the set \( \text{Val} \times \text{Val} \) of pairs of values.
(ii) The union of a family of \((G, \ell)\)-bisimulations is a \((G, \ell)\)-bisimulation. Consequently, there is a largest \((G, \ell)\)-bisimulation, which we denote \( \boxtimes_{G, \ell} \). This is the union of all such bisimulations.

One should observe that the relation \( \boxtimes_{G, \ell} \) is not reflexive. An example of an expression that is not bisimilar to itself would be \( v_\ell := ! u_\ell' \) in the context granting access right \( \kappa \) where \( \ell' \not\preceq_G \ell \) and \( \ell, \ell' \preceq_G \kappa \), which is not secure.

Our definition states that a program is secure, with respect to a default access level and a given global flow policy, if it is bisimilar to itself (as in [35]):

**Definition 3.3.3 (The Non-Disclosure Policy)** A process \( P \) satisfies the non-disclosure policy (or is secure from the confidentiality point of view) with respect to the global flow policy \( G \) if it satisfies \( P \not\boxtimes_{G, \ell} P \) for all \( \ell \). We then write \( P \in \text{ND}(G) \).

We will now define a class of expressions, denoted as “high” due to their property of not performing any changes to the low part of the memory.

**Definition 3.3.4 (Operationally High Expressions)** An expression \( M \) is said to be operationally \((F, \ell)\)-high if the following holds:

\[
\kappa \vdash_G (M, \mu) \rightarrow (M', \mu') \text{ implies } \mu \equiv_{F, \ell} \mu' \text{ and } M' \text{ is also operationally } (F, \ell)\text{-high.}
\]

All “pure expressions”, i.e. the processes that make no changes to the memory, (values, for example) are \((F, \ell)\)-high for any \( F \) and \( \ell \).

**Notation 3.3.5** The set of all operationally \((F, \ell)\)-high expressions is denoted by \( \mathcal{H}_{F, \text{low}} \).

### 3.4 Type System Properties

The following lemma establishes some properties of effects of the typable expressions that are easy to show looking at the typing rules and that will be used later in the proofs.

**Lemma 3.4.1 (Update of Effects)**

1. If \( \kappa; F; \Gamma \vdash_G E[[\text{let } u_{\ell, \theta} := V]] : s, \tau \), then \( \ell \preceq_F s.r. \)
2. If \( \kappa; F; \Gamma \vdash_G E[[\text{let } u_{\ell, \theta} := V]] : s, \tau \), then \( s.w \preceq_F \ell. \)
3. If \( \kappa; F; \Gamma \vdash_G E[[\text{ref } V]] : s, \tau \), then \( s.w \preceq_F \ell. \)
Subject Reduction

In order to establish the soundness of the type system of Figure 3.3 we need a Subject Reduction result, stating that types that are given to expressions are preserved by computation. To prove it we follow the usual steps [40] in detail.

We start by remarking that a value has no effect, and that this is properly reflected in the type system.

Moreover, the typing of a value does not depend on the current flow policy:

\textbf{Remark 3.4.2} If $W \in \text{Val}$ and $\ell; F; \Gamma \vdash_G W : s, \tau$, then for all levels $\ell'$ and flow policies $F'$, we have that $\ell'; F'; \Gamma \vdash_G W : (\bot, \top, \bot, \top, \top), \tau$.

The following lemma establishes some standard weakening and strengthening properties:

\textbf{Lemma 3.4.3}

1. If $\ell; F; \Gamma \vdash_M M : s, \tau$ and $x \notin \text{dom}(\Gamma)$, then $\ell; F; \Gamma; x : \sigma \vdash_M M : s, \tau$.

2. If $\ell; F; \Gamma; x : \sigma \vdash_M M : s, \tau$ and $x \notin \text{fv}(M)$, then $\ell; F; \Gamma \vdash_G \{ x \mapsto W \} M : s, \tau$.

\textbf{Proof:} By induction on the inference of the type judgment. \hfill \Box

We now prove two last preliminary lemmas, stating that substitutions and replacements in contexts preserve type.

\textbf{Lemma 3.4.4 (Substitution)}

If $\ell; F; \Gamma; x : \tau \vdash_M M : s, \sigma$, and by case analysis on the last rule used in this typing proof, using the previous lemma.

\textbf{NIL.} Here $\{ x \mapsto W \} M = M$, and since $x \notin \text{fv}(M)$, then by Lemma 3.4.3 we have $\ell; F; \Gamma \vdash_G M : s, \tau$.

\textbf{VAR.} If $M = x$, then $s = (\bot, \top, \bot)$, $\sigma = \tau$ and $\{ x \mapsto W \} M = W$. By Remark 3.4.2, we have $\ell; F; \Gamma \vdash_G W : (\bot, \top, \bot, \top, \top), \tau$. If $M \neq x$, then $\{ x \mapsto W \} M = M$, where $x \notin \text{fv}(M)$. Therefore, by Lemma 3.4.3, we have $\ell; F; \Gamma \vdash_G M : s, \tau$.

\textbf{ABS.} Here $M = (\lambda y.M)$, and $\ell; F; \Gamma; x : \sigma, y : \tau \vdash_M \tilde{M} : \tilde{s}, \tilde{\sigma}$ where $\tau = \tilde{\tau} = \frac{\tilde{s}}{\ell; F}$, $\tilde{\sigma}$. We can assume that $y \notin \text{dom}(\Gamma, x : \sigma)$ (otherwise rename $y$). Therefore $\{ x \mapsto W \}(\lambda y.M) = (\lambda y.(x \mapsto W)M)$. By assumption and Lemma 3.4.3 we can write $F; \Gamma, x : \sigma \vdash_M W : \sigma$. By induction hypothesis, $\ell; F; \Gamma, y : \tilde{\tau} \vdash_G \{ x \mapsto W \} M : \tilde{s}, \tilde{\sigma}$. Then, by \textbf{ABS}, $\ell; F; \Gamma \vdash_G (\lambda y.(x \mapsto W)M) : \tau$, and in particular $\ell; F; \Gamma \vdash_G (\lambda y.(x \mapsto W)M) : s, \tau$.

\textbf{REC.} Here $M = (\rho y.W)$, and $\ell; F; \Gamma; x : \sigma, y : \tau \vdash_M \tilde{W} : \tilde{s}, \tilde{\tau}$. We can assume that $y \notin \text{dom}(\Gamma, x : \sigma)$ (otherwise rename $y$). Therefore $\{ x \mapsto W \} (\rho y.W) = (\rho y.(x \mapsto W)W)$. By assumption and Lemma 3.4.3 we have $\ell; F; \Gamma, y : \tau \vdash_G W : \sigma$. By induction hypothesis, $\ell; F; \Gamma, y : \tau \vdash_G \{ x \mapsto W \} W : \tilde{s}, \tilde{\tau}$. Then, by \textbf{REC}, $\ell; F; \Gamma \vdash_G (\rho y.(x \mapsto W)W) : \tau$, and in particular we have $\ell; F; \Gamma \vdash_G (\rho y.(x \mapsto W)W) : s, \tau$.

\textbf{COND.} Here $M = \text{if } \tilde{M} \text{ then } N_1 \text{ else } N_f$ and we have $\ell; F; \Gamma, x : \sigma \vdash_G \tilde{M} : \tilde{s}, \text{bool}$, $\ell; F; \Gamma, x : \sigma \vdash_G N_1 : s_1, \tau_1$ and $\ell; F; \Gamma, x : \sigma \vdash_G N_f : s_f, \tau_2$ with $\tilde{s}, \tilde{\tau}, \tilde{\tau}_2 \subseteq \rho \{ s_1, w, s_f, w \}$ and $s = \tilde{s} \setminus s_1 \setminus s_f \setminus \{ \bot, \top, \bot, \top, \top \}$. By induction hypothesis, $\ell; F; \Gamma, x : \sigma \vdash_G \{ x \mapsto W \} M : \tilde{s}, \text{bool}$, $\ell; F; \Gamma, x : \sigma \vdash_G \{ x \mapsto W \} N_1 : s_1, \tau_1$ and $\ell; F; \Gamma, x : \sigma \vdash_G \{ x \mapsto W \} N_f : s_f, \tau_2$. Therefore, $\ell; F; \Gamma, x : \sigma \vdash_G$ if $\{ x \mapsto W \} M$ then $\{ x \mapsto W \} N_1$ else $\{ x \mapsto W \} N_f : s, \tau$ by rule \textbf{COND}.
Flow. Here \( M = \text{(flow } F \text{ in } M) \) and \( \ell; F; \Gamma, x : \sigma \vdash_{F \cup F'} M : s, \tau \). By induction hypothesis, \( \ell; F; \Gamma \vdash_{F \cup F'} \{ x \mapsto W \} M : s, \tau \). Then, by Flow, \( \ell; F; \Gamma \vdash_{F} (\text{flow } F \text{ in } \{ x \mapsto W \} M) : s, \tau \).

The proofs for the cases REF, BOOLT and BOOLF are analogous to the one for NIL, while the proofs for APP, SEQ, REF, DEREF, ENABLE, TEST and ASSIGN are analogous to the one for COND.

Lemma 3.4.5 (Replacement)
If \( \ell; F; \Gamma \vdash_{G} E[M] : s, \tau \) is a valid judgment, then the proof gives \( M \) a typing \( [E]_{\ell}; F \cup [E]; \Gamma \vdash_{G} M : s, \tau \) for some \( s \) and \( \tau \). By induction on the structure of \( \bar{s} \). We have \( \bar{s} \bar{r} \vdash \bar{r} \). Also by induction hypothesis, \( \bar{s} \bar{r} \vdash \bar{r} \). Since \( \bar{s} \bar{r} \vdash \bar{r} \), we conclude by noting that \( \bar{s} \bar{r} \vdash \bar{r} \).

Proof: By induction on the structure of \( E \).

\[ E[M] = \text{if } \bar{E}[M] \text{ then } N_{t} \text{ else } N_{f}. \]

By Cond, we have \( \ell; F; \Gamma \vdash_{G} \bar{E}[M] : \bar{s}, \text{bool} \), and \( \ell; F; \Gamma \vdash_{G} N_{t} : s_{t}, \tau \), \( \ell; F; \Gamma \vdash_{G} N_{f} : s_{f}, \tau \). By induction hypothesis, the proof gives \( M \) a typing \( \ell; F; \Gamma \vdash_{G} \bar{E}[M] : \bar{s}, \hat{\tau} \), for \( \bar{s} \). Since \( \bar{s} \bar{r} \vdash \bar{r} \), we conclude by noting that \( \bar{s} \bar{r} \vdash \bar{r} \).

We will now prove Subject Reduction, the property of type preservation by the computation, and that as the effects of an expression are performed, the security effects of the expression “weaken”. The condition \( u_{\ell, \theta} \in \text{dom}(\mu) \Rightarrow \ell; F; \Gamma \vdash_{G} \mu(u_{\ell, \theta}) : \bot, \theta \) simply says by Remark 3.4.2 that the values stored in the memory are well typed.

Theorem 3.4.6 (Subject Reduction) If \( \ell; F; \Gamma \vdash_{G} M : s, \tau \) and \( \ell; \Gamma \vdash_{G} (M, \mu) \rightarrow (M', \mu') \) with \( u_{\ell, \theta} \in \text{dom}(\mu) \Rightarrow \ell; F; \Gamma \vdash_{G} \mu(u_{\ell, \theta}) : \bot, \theta \), then \( \ell; F; \Gamma \vdash_{G} M' : s', \tau \) for some \( s' \) such that \( s' \leq_{G} s \).

Proof: We have \( M = E[R] \) and \( M' = E[N] \) with \( \ell \vdash_{G} (R, \mu) \rightarrow (N, \mu') \), where \( R \) is a redex. We proceed by induction on the context \( E \). In the case where \( E = [ ] \), the proof is as usual for the functional and imperative part of the language. Let us just examine the cases where a security construct is involved. In the cases of

\[
\begin{align*}
\ell & \vdash_{G} ((\ell \times V), \mu) \rightarrow (V, \mu) \\
\ell & \vdash_{G} ((\text{enable } \ell \text{ in } V), \mu) \rightarrow (V, \mu) \\
\ell & \vdash_{G} ((\text{flow } F \text{ in } V), \mu) \rightarrow (V, \mu)
\end{align*}
\]
we use the Lemma 3.4.2 above. The cases
\[
\ell \vdash_G ((\text{test } \ell' \text{ then } M' \text{ else } M''), \mu) \rightarrow (M', \mu) \quad \text{with } \ell' \preceq_G \ell
\]
\[
\ell \vdash_G ((\text{test } \ell' \text{ then } M'' \text{ else } M'), \mu) \rightarrow (M', \mu) \quad \text{with } \ell' \npreceq_G \ell
\]
are immediate (in the first case we use Lemma 3.2.1). Now if \( E = E'[F] \) we use the Lemma 3.4.5 above.

Now we show that the faulty expressions, as defined in Lemma 3.1.1, are not typable.

**Lemma 3.4.7** The \((\ell, G)\)-faulty expressions are not typable in the context of access right \( \ell \) and global flow policy \( G \).

**Proof:** Let \( M = E[\! u_{\ell, \theta} \!] \) with \( \ell' \npreceq_G |E|_\ell \), and assume that \( \ell; F; \Gamma \vdash_G M : s, \tau \). Then by Lemma 3.4.5 one would have \( |E|_{\ell; F \cup \{u_{\ell, \theta}\}} : s', \sigma \) for some \( s' \) and \( \sigma \), but this is only possible, by the (DEREF) rule, if \( \ell' \preceq_G |E|_\ell \), a contradiction. The other cases are standard.

An immediate consequence of these results and Lemma 3.1.1 is:

**Theorem 3.4.8 (Type Safety)** Let \( M \) be a typable closed expression, with \( \ell; F; \Gamma \vdash_G M : s, \tau \), and let \( \mu \) be such that \( u_{\ell, \theta} \in \text{dom}(\mu) \Rightarrow \ell; F; \Gamma \vdash_G \mu(u_{\ell, \theta}) : \bot, \theta \). Then either the reduction of \((M, \mu)\) with respect to \((\ell, G)\) does not terminate, or there exist a value \( V \in \text{Val} \) and a memory \( \mu' \) such that \( \ell; F; \Gamma \vdash_G (V, \mu') \) with \( \ell; F; \Gamma \vdash_G V : \bot, \tau \).

In particular, this shows that the dynamic checking of the reading clearance (by means of a “stack inspection” mechanism) is actually not needed regarding a typable program, which never attempts to access a reference for which it would not have the appropriate access right.

**Syntactically High Expressions**

We define syntactically high expressions as expressions that cannot perform changes to the low part of the memory (with respect to the current flow policy).

**Definition 3.4.9 (Syntactically “High” Expressions)** An expression \( M \) is syntactically \((F, \ell)\)-high if there exist \( \kappa, \Gamma, s, \tau \) such that \( \kappa; F; \Gamma \vdash_G M : s, \tau \) with \( s.w \npreceq_F \ell \). The expression \( M \) is a syntactically \((F, \ell)\)-high function if there exist \( \ell', F', \Gamma, s, \tau \) such that \( \ell'; F'; \Gamma \vdash M : \tau \xrightarrow{\ell, F} \sigma \) with \( s.w \npreceq_F \ell \).

It is easy to check that syntactically high expressions have an operationally high behavior.

**Lemma 3.4.10 (High Expressions)** If \( M \) is a syntactically \((F, \ell)\)-high expression, then \( M \) is an operationally \((F, \ell)\)-high expression.

### 3.4.1 Soundness

In this section we present and explain the proof of soundness of the type system of Figure 3.3 with respect to the notion of security of Definition 3.3.3. Proofs for each intermediate result are preceded by their “proof sketch”, an abridged version clarifying the intuition behind the proof.

We set to prove that, under any global flow policy \( G \), all sets of expressions \( M \) that are typable using the type system of Figure 3.3 satisfy the Non-Disclosure policy, given by Definition 3.3.3. Informally, this means that, whatever the security level that is chosen to be “low” (here that security level will be denoted by ‘low’), the expression \( M \) always presents the same behavior according to a weak bisimulation on low-equal states. We will start with an analysis of the class of expressions that are typable with a low termination effect, for which we can state a stronger soundness result.
Behavior of “Low”-Terminating Expressions

Recall that, according to the intended meaning of the termination effect, the termination or non-termination of expressions with low termination effect should only depend on the low part of the state. In other words, two computations of a same expressions running under two “low”-equal states should either both terminate or both diverge. In particular, this implies that termination-behavior of these expressions cannot be used to leak “high” information when composed with other expressions (via termination leaks). This can be formally stated as follows:

Lemma 3.4.11 (Guaranteed Transitions) Suppose that \( M \) is typable for \( \ell, F, \Gamma \). If \( \ell \vdash_G (M, \mu_1) \xrightarrow{F} (M_1', \mu_1') \) such that a reference \( a \) is fresh for \( \mu_2 \) if \( a_{\ell, \theta} \in \text{dom}(\mu_1') - \text{dom}(\mu_1) \) and we have \( \mu_1 \equiv_{F \cup F', \text{low}} \mu_2 \), then there exist \( M_2' \) and \( \mu_2' \) such that \( \ell \vdash_G (M, \mu_2) \xrightarrow{F} (M_2', \mu_2') \) with \( \mu_1' \equiv_{F, \text{low}} \mu_2' \).

We aim at proving that any typable expression \( M \) that has a low-termination effect always presents the same behavior according to a strong bisimulation on low-equal states: if two continuations \( M_1 \) and \( M_1' \) of \( M \) are related, and if \( M_1 \) can perform an execution step over a certain state, then \( M_2 \) can perform the same low changes to any low-equal state in precisely one step, while the two resulting continuations are still related. This implies that any two computations of \( M \) under low-equal states should be both finite or both infinite. To this end, we design a reflexive binary relation on expressions with low-termination effects that is closed under the transitions of Guaranteed Transitions (Lemma 3.4.11).

The definition of \( T_{G \cup F, \text{low}} \), abbreviated \( T_{F, \text{low}} \) when the global flow policy is \( G \), is given in Figure 3.4. The flow policy \( F \) is assumed to contain \( G \). Notice that it is a symmetric relation. In order to ensure that expressions that are related by \( T_{F, \text{low}} \) perform the same changes to the low memory, its definition requires that the references that are created or written using (potentially) different values are high.

Remark 3.4.13 If for some \( F \) we have that \( M_1 T_{F, \text{low}} M_2 \) and \( M_1 \in \text{Val} \), then \( M_2 \in \text{Val} \).

We have seen in Splitting Computations (Lemma 3.1.2) that two computations of the same expression can split only if the expression is about to read a reference that is given different values by the memories in each of the configurations. Since we will be only interested in the case where the two memories are low-equal, this situation coincides with the case where the reference that is read is high. From the following lemma one can conclude that the relation \( T_{F, \text{low}} \) relates the possible outcomes of expressions that are typable with a low termination effect, and that perform a high read over low-equal memories.

Lemma 3.4.14 If there exist \( \kappa, \Gamma, s, \tau \) such that \( \kappa; F; \Gamma \vdash_G E[[a_{\ell, \theta}]] : s.\tau \) with \( s.t \equiv_F \text{low} \) and \( j \not\equiv_{F \cup \text{low}[E]} \text{low} \), then for any values \( V_0, V_1 \in \text{Val} \) such that \( \kappa; F; \Gamma \vdash_G V_i : \theta \) we have \( E[V_0] T_{F, \text{low}} E[V_1] \).

Proof sketch. If a typable expression is about to use a value that results from a high dereference in such a way that it could influence its termination behavior, then its termination effect cannot be low (contradicting the assumption). The type system enforces this by updating the termination effect of the expression with the reading effect of the dereferencing operation, in the cases where the value is used: in the predicate of a conditional (\( s.r \) in the termination effect of \( \text{COND} \)); to determine the function of an application (\( s.r \) in the termination effect of \( \text{APP} \)); to determine the argument of an application (\( s''.r \) in the termination effect of \( \text{APP} \)) of an application.

The relation \( T \) requires that the references that are (respectively) created or written using the high dereferenced value are high (see Clauses 4 and 6). This is guaranteed by conditions of the form \( \langle s.r \leq_F \ell \rangle \), where \( s \) is the security effect of the program that is performing the access, and \( j \) is the security level of the reference that is created or written. More precisely, conditions are imposed when the dereferenced value is used: to create a reference (\( s.r \leq_F \ell \) in rule \( \text{REF} \)); to determine a reference that is being assigned to (\( s.r \leq_F \ell \) in rule \( \text{ASSIGN} \)); to determine a value that is being assigned (\( s'.r \leq_F \ell \) in rule \( \text{ASSIGN} \)).
Definition 3.4.12 ($T_{F,\text{low}}$)
We have that $M_1 T_{F,\text{low}} M_2$ if $\kappa; F; \Gamma \vdash_G M_1 : s_1, \tau$ and $\kappa; F; \Gamma \vdash_G M_2 : s_2, \tau$ for some $\kappa, F, \Gamma, s_1, s_2$ and $\tau$ with $s_1.t \leq_F \text{low}$ and $s_2.t \leq_F \text{low}$ and one of the following holds:

Clause 1. $M_1$ and $M_2$ are both values, or

Clause 2. $M_1 = M_2$, or

Clause 3. $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1 T_{F,\text{low}} \bar{M}_2$, or

Clause 4. $M_1 = (\text{ref}_{\kappa, \theta} \bar{M}_1)$ and $M_2 = (\text{ref}_{\kappa, \theta} \bar{M}_2)$ with $\bar{M}_1 T_{F,\text{low}} \bar{M}_2$, and $j \not\in_F \text{low}$, or

Clause 5. $M_1 = (\text{!} M_1)$ and $M_2 = (\text{!} \bar{M}_2)$ with $M_1 T_{F,\text{low}} \bar{M}_2$, or

Clause 6. $M_1 = (M_1 := \bar{N}_1)$ and $M_2 = (M_2 := \bar{N}_2)$ with $M_1 T_{F,\text{low}} \bar{M}_2$, and $\bar{N}_1 T_{F,\text{low}} \bar{N}_2$, and $M_1, \bar{M}_2$ both have type $\theta$ ref$_j$ for some $\theta$, and $l$ such that $j \not\in_F \text{low}$, or

Clause 7. $M_1 = (\text{flow} F' \text{ in } M_1)$ and $M_2 = (\text{flow} F' \text{ in } \bar{M}_2)$ with $M_1 T_{F', F', \text{low}} \bar{M}_2$

Clause 8. $M_1 = (\tau \times M_1)$ and $M_2 = (\tau \times M_2)$ with $M_1 T_{F, \text{low}} \bar{M}_2$.

Clause 9. $M_1 = (\text{enable } i \text{ in } M_1)$ and $M_2 = (\text{enable } i \text{ in } M_2)$ with $M_1 T_{F,\text{low}} \bar{M}_2$.

Figure 3.4: The relation $T_{F,\text{low}}$

Proof: By induction on the structure of E.

E[(! a_{j, \theta})] = (! a_{j, \theta}). We have $V_0 T_{F,\text{low}} V_1$ by Clause 1.

E[(! a_{j, \theta})] = (E_1[(! a_{j, \theta})] M). By App we have $\kappa; F; \Gamma \vdash_G E_1[(! a_{j, \theta})] : \bar{s}, \bar{\tau} \leftarrow \bar{s}, \bar{\tau}$ with $\bar{s}.r \leq_F \text{low}$. By Lemma 3.4.1, we have $j \leq_F \bar{s}.r$. Therefore $j \leq_F \bar{s}.r$, which contradicts the assumption that both $s.t \leq_F \text{low}$ and $j \not\in_F \text{low}$ hold.

E[(! a_{j, \theta})] = (V E_1[(! a_{j, \theta})]). By rule App we have $\kappa; F; \Gamma \vdash_G E_1[(! a_{j, \theta})] : \bar{s}, \bar{\tau} \leftarrow \bar{s}, \bar{\tau}$ with $\bar{s}.r \leq_F \text{low}$. By Lemma 3.4.1, we have $j \leq_F \bar{s}.r$. Therefore $j \leq_F \bar{s}.r$, which contradicts the assumption that both $s.t \leq_F \text{low}$ and $j \not\in_F \text{low}$ hold.

E[(! a_{j, \theta})] = if E_1[(! a_{j, \theta})] then M_1 else M_2. By Cond we have that $\kappa; F; \Gamma \vdash_G E_1[(! a_{j, \theta})] : \bar{s}, \bar{\tau}$ with $\bar{s}.r \leq_F \text{low}$. By Lemma 3.4.1, we have $j \leq_F \bar{s}.r$. Therefore $j \leq_F \bar{s}.r$, which contradicts the assumption that both $s.t \leq_F \text{low}$ and $j \not\in_F \text{low}$ hold.

E[(! a_{j, \theta})] = (E_1[(! a_{j, \theta})]; M). By seq we have $\kappa; F; \Gamma \vdash_G E_1[(! a_{j, \theta})] : \bar{s}, \bar{\tau}$ with $\bar{s}.t \leq_F \text{low}$. Therefore $\bar{s}.t \leq_F \text{low}$, and since $l \not\in_F \text{low}$ implies $l \not\in_F \text{low}$, then by induction hypothesis we have $E_1[V_0] T_{F,\text{low}} E_1[V_1]$. By Lemma 3.4.5 and Clause 3 we can conclude.

E[(! a_{j, \theta})] = (\text{ref}_{\kappa, \theta} E_1[(! a_{j, \theta})]). By rule Ref we have that $\kappa; F; \Gamma \vdash_G E_1[(! a_{j, \theta})] : \bar{s}, \bar{\tau}$ with $\bar{s}.r = \bar{s}.t \leq_F \text{low}$. Therefore $\bar{s}.t \leq_F \text{low}$, and since $l \not\in_F \text{low}$ implies $j \not\in_F \text{low}$, then by induction hypothesis we have $E_1[V_0] T_{F,\text{low}} E_1[V_1]$ by Lemma 3.4.1 we have $j \leq_F \bar{s}.r$, so $s.r \not\in_F \text{low}$. Therefore, $l' \not\in_F \text{low}$, and we conclude by Lemma 3.4.5 and Clause 4.
E[(! a_{j,\theta})] = (! E_1[(! a_{j,\theta})]). By rule DEREF we have $\kappa; F; \Gamma \vdash_G E_1[(! a_{j,\theta})] : \bar{s}.\bar{r}$ with $\bar{s}.t \preceq_F s.t$. Therefore $\bar{s}.t \preceq_F \ell$, low, and since $j \not\preceq_{F \cup \{ E_1 \}} \ell$ low implies $l \not\preceq_{F \cup \{ E_1 \}} \ell$, low, then by induction hypothesis $E_1[V_0] \quad T_{F,low} \ E_1[V_1]$. We conclude by Lemma 3.4.5 and Clause 5.

E[(! a_{j,\theta})] = (E_1[(! a_{j,\theta})]) := M). By rule ASSIGN we have that $\kappa; F; \Gamma \vdash_G E_1[a_{j,\theta}] : \bar{s},\bar{\theta}$ ref with $\bar{s}.t \preceq_F s.t$ and $\bar{s}.r \preceq_F \ell$. Therefore $\bar{s}.t \preceq_F \ell$, low, and since $j \not\preceq_{F \cup \{ E_1 \}} \ell$ low implies $j \not\preceq_{F \cup \{ E_1 \}} \ell$, low, then by induction hypothesis $E_1[V_0] \quad T_{F,low} \ E_1[V_1]$. On the other hand, by Clause 2 we have $M \quad T_{F,low} \ M$. By Lemma 3.4.1 we have $j \preceq_F \bar{s}.r$, so $j \preceq_F \ell$. Then, we must have $\ell \not\preceq_F \ell$, since otherwise $j \preceq_{F \cup \{ E_1 \}} \ell$. Therefore, we conclude by Lemma 3.4.5 and Clause 6.

E[(! a_{j,\theta})] = (V := E_1[(! a_{j,\theta})]). By rule ASSIGN we have that $\kappa; F; \Gamma \vdash_G V : \bar{s},\bar{\theta}$ ref, and $\ell; F; \Gamma \vdash_G E_1[a_{j,\theta}] : \bar{s}'.\bar{\theta}$ with $\bar{s}'.t \preceq_F s.t$ and $\bar{s}'.r \preceq_F \ell$. Therefore $\bar{s}'.t \preceq_F \ell$, low, and since $j \not\preceq_{F \cup \{ E_1 \}} \ell$ low implies $j \not\preceq_{F \cup \{ E_1 \}} \ell$, low, then by induction hypothesis $E_1[V_0] \quad T_{F,low} \ E_1[V_1]$. On the other hand, by Clause 2 we have $V \quad T_{F,low} \ V$. By Lemma 3.4.1 we have $l \preceq_F \bar{s}.r$, so $j \preceq_F \ell$. Then, we must have $\ell \not\preceq_F \ell$, since otherwise $j \preceq_{F \cup \{ E_1 \}} \ell$. We then conclude by Lemma 3.4.5 and Clause 6.

E[(! a_{j,\theta})] = (flow F' in E_1[(! a_{j,\theta})]). By rule FLOW we have $\kappa; F \cup F'; \Gamma \vdash_G V : s,\tau$. By induction hypothesis $E_1[V_0] \quad T_{F \cup F',low} \ E_1[V_1]$, so we conclude by Lemma 3.4.5 and Clause 7.

E[(! a_{j,\theta})] = (a \times E_1[(! a_{j,\theta})]). By rule RESTRICT we have $\kappa; \lambda; \Gamma \vdash_G V : s,\tau$. By induction hypothesis $E_1[V_0] \quad T_{F,low} \ E_1[V_1]$, so we conclude by Lemma 3.4.5 and Clause 8.

E[(! a_{j,\theta})] = (enable e in E_1[(! a_{j,\theta})]). By rule ENABLE we have $\kappa; \Gamma \vdash_G V : s,\tau$. By induction hypothesis $E_1[V_0] \quad T_{F \cup F',low} \ E_1[V_1]$, so we conclude by Lemma 3.4.5 and Clause 9.

We can now prove that $T_{F,low}$ behaves as a kind of “strong bisimulation”:

**Theorem 3.4.15 (Strong Bisimulation for Low-Terminating Expressions)**

If we have $M_1 \quad T_{F,low} \ M_2$ and $\ell \vdash_G (M_1,\mu_1) \xrightarrow{\ell} (M_1',\mu_1')$, with $\mu_1 \preceq_{F \cup F',low} \mu_2$ such that $a$ is fresh for $\mu_2$ if $a_{j,\theta} \in \text{dom}(\mu_1') - \text{dom}(\mu_1)$, then there exist $M_2'$ and $\mu_2'$ such that $\ell \vdash_G (M_2,\mu_2) \xrightarrow{\ell} (M_2',\mu_2')$ with $M_1' \quad T_{F,low} \ M_2'$ and $\mu_1' \preceq_{F,low} \mu_2'$.

**Proof sketch.** If $M_1$ and $M_2$ are equal (related by $T$ using Clause 2), then since they have a low termination effect we can use Guaranteed Transitions (Lemma 3.4.11) to conclude that $M_2$ can also make a step and perform the same changes to low-equal memories. If the result of performing the two steps is different – therefore not falling again in Clause 2 – by Splitting Computations (Lemma 3.1.2) we conclude they have performed a high dereference. In Lemma 3.4.14 we have seen that this implies that the resulting expressions are still in the $T$ relation.

The remaining cases use the fact that $M_1$ is a value if and only if $M_2$ is a value, to show that if $M_1$ can perform a computation step, so can $M_2$.

**Proof:** By induction on the definition of $T_{F,low}$. In the following, we use Subject Reduction (Theorem 3.4.6) to guarantee that the termination effect of the expressions resulting from $M_1$ and $M_2$ is still low with respect to low and $F$. This, as well as typability (with the same type) for low and $F$, is a requirement for being in the $T_{F,low}$ relation.

**Clause 1.** This case is not possible.
Clause 2. Here $M_1 = M_2$. By Guaranteed Transitions (Lemma 3.4.11) there exist $M'_2$ and $\mu'_2$ such that $\ell \vdash_G (M_2, \mu_1) \rightarrow (M'_2, \mu'_2)$ with $\mu'_1 \not\preceq \ell \vdash_G (M_2, \mu_2)$.

$M'_2 = M'_1$. Then we have $M'_1 \vdash F, \ell \vdash_G (M'_2, \mu'_2)$, by Clause 2 and Subject Reduction (Theorem 3.4.6).

$M'_2 \not\equiv M'_1$. Then by Splitting Computations (Lemma 3.1.2) we have that there exist $E$ and $a_{j, \theta}$ such that $E'[1] = [E]$, $M'_1 = E[\mu_1(a_{j, \theta})]$, $M'_2 = E[\mu_2(a_{j, \theta})]$, $\mu'_1 = \mu_1$ and $\mu'_2 = \mu_2$. Since $\mu_1(a_{j, \theta}) \not\preceq \mu_2(a_{j, \theta})$, we have $j \not\preceq _F \ell \vdash_G (M'_2, \mu'_2)$, by Lemma 3.4.14 above.

Clause 3. Here $M_1 = (M_1; \bar{N})$ and $M_2 = (M_2; \bar{N})$ where $M_1 \vdash F, \ell \vdash_G (M'_2, \mu'_2)$. Then either:

$\bar{M}_1$ can compute. In this case $M'_1 = (\bar{M}_1' \supset N)$ with $\ell \vdash_G (\bar{M}_1, \mu_1) \rightarrow (\bar{M}_1', \mu'_1)$. We use the induction hypothesis, Clause 3 and Subject Reduction (Theorem 3.4.6) to conclude.

$\bar{M}_1$ is a value. In this case $M'_1 = \bar{N}$ and $E' = \emptyset$ and $\mu'_1 = \mu_1$. We have $\bar{M}_2 \in \text{Val}$ by Remark 3.4.13, hence $\ell \vdash_G (M_2, \mu_2) \rightarrow (\bar{N}, \mu_2)$, and we conclude using Clause 2 and Subject Reduction (Theorem 3.4.6).

Clause 4. Here $M_1 = (\text{ref}, \theta \bar{M}_1)$ and $M_2 = (\text{ref}, \theta \bar{M}_2)$ where $M_1 \vdash F, \ell \vdash_G (\bar{M}_2, \mu_2)$, $\ell \not\preceq F, \ell \vdash_G (M_2, \mu_2)$. There are two cases.

$\bar{M}_1$ can compute. In this case $M'_1 = (\bar{M}_1' \supset \bar{M}_1)$ with $\ell \vdash_G (\bar{M}_1, \mu_1) \rightarrow (\bar{M}_1', \mu'_1)$. We use the induction hypothesis, Subject Reduction (Theorem 3.4.6) and Clause 4 to conclude.

$\bar{M}_1$ is a value. In this case $M'_1 = a_{j, \theta}$, with $a$ fresh for $\mu_1$, $F' = \emptyset$ and $\mu'_1 = \mu_1 \cup \{a_{j, \theta} \rightarrow \bar{M}_1\}$ (and therefore $a$ is also fresh for $\mu_2$). Then $\bar{M}_2 \in \text{Val}$ by Remark 3.4.13, and therefore $\ell \vdash_G (M_2, \mu_2) \rightarrow (M_2, \mu_2)$, we get $\mu'_1 \not\preceq F, \ell \vdash_G (M_2, \mu_2)$ since $\ell \not\preceq F, \ell \vdash_G (M_2, \mu_2)$. We conclude using Clause 1 and Subject Reduction (Theorem 3.4.6).

Clause 5. Here $M_1 = (! \bar{M}_1)$ and $M_2 = (! \bar{M}_2)$ where $M_1 \vdash F, \ell \vdash_G (\bar{M}_2, \mu_2)$. We distinguish two sub-cases.

$\bar{M}_1$ can compute. In this case $\ell \vdash_G (\bar{M}_1, \mu_1) \rightarrow (\bar{M}_1', \mu'_1)$. We use the induction hypothesis, Subject Reduction (Theorem 3.4.6) and Clause 5 to conclude.

$\bar{M}_1$ is a value. Then $\bar{M}_1 = u_{j, \theta}$ and $M'_1 \in \text{Val}$, and $\mu'_1 = \mu_1$, $F' = \emptyset$. By Remark 3.4.13, $\bar{M}_2 \in \text{Val}$, and since $M_1$ and $M_2$ have the same type, it must be a reference $v_{j, \theta}$, and so $\ell \vdash_G (M_2, \mu_2) \rightarrow (v_{j, \theta}, \mu_2)$. We then conclude using Clause 1 and Subject Reduction (Theorem 3.4.6).

Clause 6. Here we have $M_1 = (\bar{M}_1 := \bar{N}_1)$ and $M_2 = (\bar{M}_2 := \bar{N}_2)$ where $\bar{M}_1 \vdash F, \ell \vdash_G (\bar{N}_1, \mu_1)$ and $\bar{M}_2 \vdash F, \ell \vdash_G (\bar{N}_2, \mu_2)$.

$\bar{M}_1$ can compute. In this case $\ell \vdash_G (\bar{M}_1, \mu_1) \rightarrow (\bar{M}_1', \mu'_1)$. We use the induction hypothesis, Subject Reduction (Theorem 3.4.6) and Clause 6 to conclude.

$\bar{M}_1$ is a value, but $\bar{N}_1$ can compute. In this case we have $\ell \vdash_G (\bar{N}_1, \mu_1) \rightarrow (\bar{N}_1', \mu'_1)$. By Remark 3.4.13 also $\bar{M}_2 \in \text{Val}$. We use the induction hypothesis, Subject Reduction (Theorem 3.4.6) and Clause 6 to conclude.

$\bar{M}_1$ and $\bar{N}_1$ are values. Then $\bar{M}_1 = u_{j, \theta}$ and $M'_1 = \emptyset$, $\mu'_1 = \{\bar{M}_1 \rightarrow \bar{M}_1\} \mu_1$, $F' = \emptyset$. By Remark 3.4.13, also $\bar{M}_2, \bar{N}_2 \in \text{Val}$, and since $M_1$ and $M_2$ have the same type, $\bar{M}_2$ must be a reference $v_{j, \theta}$, and so $\ell \vdash_G (M_2, \mu_2) \rightarrow (\emptyset, \{\bar{M}_2 \rightarrow \bar{M}_2\} \mu_2)$. Since $\ell \vdash_G (M_2, \mu_2) \rightarrow (\emptyset, \{\bar{N}_2 \rightarrow \bar{M}_2\} \mu_2)$. We then conclude using Clause 1 and Subject Reduction (Theorem 3.4.6).


Clause 7. Here we have $M_1 = (\text{flow } F \text{ in } \bar{M}_1)$ and $M_2 = (\text{flow } F \text{ in } \bar{M}_2)$ and $\bar{M}_1 \mathcal{T}_{F, \text{low}} \bar{M}_2$. There are two cases.

$\bar{M}_1 \text{ can compute.}$ In this case $\ell \vdash_G (\bar{M}_1, \mu_1) \xrightarrow{F'} (\bar{M}_1', \mu_1')$ with $F' = \bar{F} \cup F''$. By induction hypothesis, we have $\ell \vdash_G (M_2, \mu_2) \xrightarrow{F'} (M_2', \mu_2')$, and $M_1' \mathcal{T}_{F, \text{low}} M_2'$ and $\mu_1' \equiv_{F \cup F', \text{low}} \mu_2'$. Notice that this implies $M_1' \equiv_{F, \text{low}} \mu_2'$. We use Subject Reduction (Theorem 3.4.6) and Clause 7 to conclude.

$\bar{M}_1 \text{ is a value.}$ In this case $M_1' = \bar{M}_1$, $F' = \emptyset$ and $\mu_1' = \mu_1$. Then $\bar{M}_2 \in \text{Val}$ by Remark 3.4.13, and so $\ell \vdash_G (M_2, \mu_2) \xrightarrow{F'} (\bar{M}_2, \mu_2)$. We conclude using Clause 1 and Subject Reduction (Theorem 3.4.6).

Clause 8. Here we have $M_1 = (\ell \times \bar{M}_1)$ and $M_2 = (\ell \times \bar{M}_2)$ and $\bar{M}_1 \mathcal{T}_{F, \text{low}} \bar{M}_2$. There are two cases.

$\bar{M}_1 \text{ can compute.}$ In this case $\ell \vdash_G (\bar{M}_1, \mu_1) \xrightarrow{F'} (\bar{M}_1', \mu_1')$. We use the induction hypothesis, Subject Reduction (Theorem 3.4.6) and Clause 8 to conclude.

$\bar{M}_1 \text{ is a value.}$ In this case $M_1' = \bar{M}_1$, $F' = \emptyset$ and $\mu_1' = \mu_1$. Then $\bar{M}_2 \in \text{Val}$ by Remark 3.4.13, and so $\ell \vdash_G (M_2, \mu_2) \xrightarrow{F'} (\bar{M}_2, \mu_2)$. We conclude using Clause 1 and Subject Reduction (Theorem 3.4.6).

Clause 9. Here we have $M_1 = (\text{enable } \ell \text{ in } \bar{M}_1)$ and $M_2 = (\text{enable } \ell \text{ in } \bar{M}_2)$ and $\bar{M}_1 \mathcal{T}_{F, \text{low}} \bar{M}_2$. There are two cases.

$\bar{M}_1 \text{ can compute.}$ In this case $\ell \vdash_G (\bar{M}_1, \mu_1) \xrightarrow{F'} (\bar{M}_1', \mu_1')$. We use the induction hypothesis, Subject Reduction (Theorem 3.4.6) and Clause 9 to conclude.

$\bar{M}_1 \text{ is a value.}$ In this case $M_1' = \bar{M}_1$, $F' = \emptyset$ and $\mu_1' = \mu_1$. Then $\bar{M}_2 \in \text{Val}$ by Remark 3.4.13, and so $\ell \vdash_G (M_2, \mu_2) \xrightarrow{F'} (\bar{M}_2, \mu_2)$. We conclude using Clause 1 and Subject Reduction (Theorem 3.4.6).

We have seen in Remark 3.4.13 that when two expressions are related by $\mathcal{T}_{F, \text{low}}$ and one of them is a value, then the other one is also a value. From a semantical point of view, when an expression has reached a value it means that it has successfully completed its computation. We will now see that when two expressions are related by $\mathcal{T}_{F, \text{low}}$ and one of them is unable to resolve into a value, in any sequence of unrelated computation steps, then the other one is also unable to do so.

Definition 3.4.16 (Non-resolvable Expressions) We say that an expression $M$ is non-resolvable, denoted $M \dagger$, if there is no derivative $M'$ of $M$ such that $M' \in \text{Val}$.

Lemma 3.4.17 If for some $F$ we have that $M \mathcal{T}_{F, \text{low}} N$ and $M \dagger$, then $N \dagger$.

Proof sketch. We prove that if $M$ and $N$ are related by $\mathcal{T}$, and there is a sequence of memories that defines a path of execution steps from $N$ into a value, then the sequence of memories can be used to “bring” $M$ into a value as well. This can be seen using Strong Bisimulation for Low-Terminating Expressions (Proposition 3.4.15), since for each step between two derivatives of $N$ it guarantees a step between the corresponding two derivatives of $M$, in such a way that the relation $\mathcal{T}$ is maintained.
**Proof:** Let us suppose that \( \neg N^\uparrow \). That means that there exists a finite number of states \( \mu_1, \ldots, \mu_n \) and \( \mu'_1, \ldots, \mu'_n \) and of expressions \( N_1, \ldots, N_n \) such that

\[
\ell \vdash_G (N, \mu_1) \rightarrow (N_1, \mu'_1) \quad \text{and} \\
\ell \vdash_G (N, \mu_2) \rightarrow (N_2, \mu'_2) \quad \text{and} \\
\vdots \\
\ell \vdash_G (N_{n-1}, \mu_n) \rightarrow (N_n, \mu'_n)
\]

and such that \( N_n \in \text{Val} \). By Strong Bisimulation for Low-Terminating Threads (Proposition 3.4.15), we have that there exists a finite number of states \( \bar{\mu}'_1, \ldots, \bar{\mu}'_n \) and of expressions \( M_1, \ldots, M_n \) such that

\[
\ell \vdash_G (M, \mu_1) \rightarrow (M_1, \bar{\mu}'_1) \quad \text{and} \\
\ell \vdash_G (M, \mu_2) \rightarrow (M_2, \bar{\mu}'_2) \quad \text{and} \\
\vdots \\
\ell \vdash_G (M_{n-1}, \mu_n) \rightarrow (M_n, \bar{\mu}'_n)
\]

such that:

\[M_1 \uparrow_{F,\text{low}} N_1, \text{ and } \ldots, \text{ and } M_n \uparrow_{F,\text{low}} N_n\]

By Remark 3.4.13, we then have that \( M_n \in \text{Val} \). Since \( M_n \) is a derivative of \( M \), we conclude that \( \neg M^\uparrow \). \( \square \)

The following lemma deduces operational "highness" of expressions from that of its subexpressions.

**Lemma 3.4.18 (Composition of High Expressions)** Suppose that \( M \) is typable in \( \kappa \) and \( F \). Then:

1. If \( M = (M_1 \cdot M_2) \) and \( M_1 \) is a syntactically \((F,\text{low})\)-high function and either
   - \( M_1^\uparrow \) and \( M_1 \in \mathcal{H}_{F,\text{low}} \), or
   - \( M_1, M_2 \in \mathcal{H}_{F,\text{low}} \)
   then \( M \in \mathcal{H}_{F,\text{low}} \).

2. If \( M = \text{if } M_l \text{ then } M_f \text{ else } M_i \) and \( M_1, M_f \in \mathcal{H}_{F,\text{low}} \), then \( M \in \mathcal{H}_{F,\text{low}} \).

3. If \( M = (\text{ref}_j \theta M_1) \) and \( j \notin F \) low and \( M_1 \in \mathcal{H}_{F,\text{low}} \), then \( M \in \mathcal{H}_{F,\text{low}} \).

4. If \( M = (M_1 :: M_2) \) and either
   - \( M_1^\uparrow \) and \( M_1 \in \mathcal{H}_{F,\text{low}} \), or
   - \( M_1, M_2 \in \mathcal{H}_{F,\text{low}} \)
   then \( M \in \mathcal{H}_{F,\text{low}} \).

5. If \( M = (M_1 := M_2) \) and \( M_1 \) has type \( \theta \) \( \text{ref}_j \) with \( j \notin F \) low and either
   - \( M_1^\uparrow \) and \( M_1 \in \mathcal{H}_{F,\text{low}} \), or
   - \( M_1, M_2 \in \mathcal{H}_{F,\text{low}} \)
   then \( M \in \mathcal{H}_{F,\text{low}} \).

6. If \( M = \text{(flow } F' \text{ in } M_1) \) and \( M_1 \in \mathcal{H}_{F \cup F',\text{low}} \), then \( M \in \mathcal{H}_{F,\text{low}} \).
7. If \( M = (i \times M_1) \) and \( M_1 \in \mathcal{H}_{F,low} \), then \( M \in \mathcal{H}_{F,low} \).

8. If \( M = (\text{enable } i \text{ in } M_1) \) and \( M_1 \in \mathcal{H}_{F,low} \), then \( M \in \mathcal{H}_{F,low} \).

**Proof sketch.** A construct that does not introduce low effects and that is only composed of operationally high expressions can be easily seen to be operationally high: for all the computation steps that can be performed by any of its derivatives, there is a corresponding one that can be performed by a derivative of one of its components. Since the components are operationally high, then the step does not make low changes to the state.

Syntactical highness of a function guarantees that its body, which can be seen as a subexpression of an application, is operationally high. A reference creation or assignment that is only composed of operationally high expressions is operationally high for the same reasons, provided that the created or written reference is high.

When a non-resolvable expression \( M_1 \) is composed with an expression \( M_2 \), as in \((M_1 \ M_2)\), \((M_1 ; M_2)\) or \((M_1 := M_2)\), it is enough to require that \( M_1 \) is operationally high. In fact, for all the computation steps that can be performed by any of these expressions' derivatives, there is a corresponding one that can be performed by a derivative of \( M_1 \) — that is, the expression \( M_2 \) never gets to be evaluated.

**Proof:** We give the proof for the case where \( M = (M_1 \ M_2) \) and \( M_1 \) is a syntactically \((F,\text{low})\)-high function. The other cases are analogous or simpler.

**\( M_1 \dagger \) and \( M_1 \in \mathcal{H}_{F,low} \).** Let \( \mathcal{F} \) be a set of expressions that includes \( \mathcal{H}_{F,low} \), and that contains the expressions \((M_1 \ M_2)\) provided that they are typable in \( F \), and satisfy \( M_1 \not\in \text{Val} \) and \( M_1 \in \mathcal{F} \) and \( M_1 \) is a \((F,\text{low})\)-high function. Assume that an application \( M = (M_1 \ M_2) \) such that \( M_1 \dagger \) and \( M_1 \in \mathcal{H}_{F,low} \) performs the transition \( \ell \vdash_G (M, \mu) \rightarrow^F (M', \mu') \). We show that this implies \( M' \in \mathcal{F} \) and \( \mu \simeq_{F,\text{low}} \mu' \).

Since \( M_1 \) is non-resolvable, \( M_1 \) cannot be a value, and since \( M \) can compute, then also \( M_1 \) can compute. We then have \( M' = (M'_1 \ M_2) \) with \( \ell \vdash_G (M_1, \mu) \rightarrow^F (M'_1, \mu') \). Since \( M_1 \in \mathcal{H}_{F,low} \), then also \( M'_1 \in \mathcal{H}_{F,low} \), thus \( M'_1 \in \mathcal{F} \), and \( \mu \simeq_{F,\text{low}} \mu' \). By Subject Reduction (Theorem 3.4.6), \( M'_1 \) is a \((F,\text{low})\)-high function, and since \( M_1 \dagger \), then \( M'_1 \not\in \text{Val} \). Hence \( M' \in \mathcal{F} \).

**\( M_1, M_2 \in \mathcal{H}_{F,low} \).** Let \( \mathcal{F} \) be a set of expressions that includes \( \mathcal{H}_{F,low} \), and that contains expressions \((M_1 \ M_2)\) provided they are typable in \( F \) and satisfy \( M_1, M_2 \in \mathcal{F} \) and \( M_1 \) is a \((F,\text{low})\)-high function. Assume that an expression \( M = (M_1 \ M_2) \) is a \((F,\text{low})\)-high function, and by Substitution (Lemma 3.4.4), also \( M' \) is a \((F,\text{low})\)-high function. Therefore, by High Expressions (Lemma 3.4.10), \( M' \in \mathcal{H}_{F,low} \).

**\( M_1 \) and \( M_2 \) are values.** Then \( M_1 = (\lambda x. M_1) \) \( M' = \{ x \mapsto M_2 \} \bar{M}_1 \) and \( \mu = \mu' \). Since \( M_1 \) is a \((F,\text{low})\)-high function, then by Abs \( M_1 \) is syntactically \((F,\text{low})\)-high, and by Substitution (Lemma 3.4.4), also \( M' \) is syntactically \((F,\text{low})\)-high. Therefore, by High Expressions (Lemma 3.4.10), \( M' \in \mathcal{H}_{F,low} \).

**\( M_1 \) can compute.** Then we have \( M' = (M'_1 \ M_2) \) with \( \ell \vdash_G (M_1, \mu) \rightarrow^F (M'_1, \mu') \). Since \( M_1 \in \mathcal{H}_{F,low} \), then also \( M'_1 \in \mathcal{F} \) and \( \mu \simeq_{F,\text{low}} \mu' \). By Subject Reduction (Theorem 3.4.6) \( M'_1 \) is a \((F,\text{low})\)-high function. Hence \( M' \in \mathcal{F} \).

**\( M_1 \) is a value but \( M_2 \) can compute.** Then we have \( M' = (M_1 \ M'_2) \) with \( \ell \vdash_G (M_2, \mu) \rightarrow^F (M'_2, \mu') \). Since \( M_2 \in \mathcal{H}_{F,low} \), then also \( M'_2 \in \mathcal{F} \) and \( \mu \simeq_{F,\text{low}} \mu' \). Hence \( M' \in \mathcal{F} \).

**Lemma 3.4.19** If for some \( F \) we have that \( M_1 \in \mathcal{H}_{F,low} \) and \( M_1 \in \mathcal{H}_{F,low} \), then \( M_2 \in \mathcal{H}_{F,low} \).
3.4 Type System Properties

**Proof sketch.** The proof relies on the fact that if an expression $M_1$ of the form $(\tilde{M}_1; \tilde{N}_1)$ or $(\tilde{M}_1 := \tilde{N}_1)$ is operationally high, in spite of $\tilde{N}_1$ not being operationally high, then $\tilde{M}_1$ is non-resolvable. To see this, note that if $M_1$ were not non-resolvable, we would have, for some value $V$, that $(V; \tilde{N}_1)$ or $(V := \tilde{N}_1)$ would be derivatives of $M_1$. We can then see that, for all the computation steps that can be performed by any of $\tilde{N}_1$'s derivatives, there is a corresponding one that can be performed by a derivative of $M_1$. Since $\tilde{N}_1$ is not operationally high, then also $M$ would not be operationally high.

From the fact that an expression is operationally high, we can easily conclude that the first subexpression to be evaluated is also operationally high. Clauses 3 and 6 do not require their second subexpression $\tilde{N}_1$ to be operationally high. However, by the above observation and by Lemma 3.4.17 this implies that $\tilde{M}_2$ is non-resolvable. We can then argue that the expressions in the $T$ relation have the same “composition”, and conclude that they are operationally high using Composition of High Expressions (Lemma 3.4.18).

**Proof:** By induction on the definition of $M_1 \ T_{F,low} M_2$.

**Clause 1.** Direct.

**Clause 2.** Direct.

**Clause 3.** Here $M_1 = (\tilde{M}_1; \tilde{N})$ and $M_2 = (\tilde{M}_2; \tilde{N})$ with $\tilde{M}_1 \ T_{F,low} \tilde{M}_2$. Clearly we have that $\tilde{M}_1 \in H_{F,low}$, so by induction hypothesis, also $\tilde{M}_2 \in H_{F,low}$. We distinguish two sub-cases:

- $\tilde{N} \in H_{F,low}$. Then, $\tilde{M}_2, \tilde{N} \in H_{F,low}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,low}$.
- $\tilde{N} \notin H_{F,low}$. Then $\tilde{M}_1 \dagger$, and by Lemma 3.4.17 also $\tilde{M}_2 \dagger$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,low}$.

**Clause 4.** Here $M_1 = (\text{ref}_{j,\theta} \tilde{M}_1)$ and $M_2 = (\text{ref}_{j,\theta} \tilde{M}_2)$ where $\tilde{M}_1 \ T_{F,low} \tilde{M}_2$, and $j \notin F$ low. Clearly we have that $\tilde{M}_1 \in H_{F,low}$, so by induction hypothesis also $\tilde{M}_2 \in H_{F,low}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,low}$.

**Clause 5.** Here $M_1 = (! \tilde{M}_1)$ and $M_2 = (! \tilde{M}_2)$ where $\tilde{M}_1 \ T_{F,low} \tilde{M}_2$. Clearly we have that $\tilde{M}_1 \in H_{F,low}$, so by induction hypothesis also $\tilde{M}_2 \in H_{F,low}$. This implies that $M_2 \in H_{F,low}$.

**Clause 6.** Here we have $M_1 = (\tilde{M}_1 := \tilde{N}_1)$ and $M_2 = (\tilde{M}_2 := \tilde{N}_2)$ where $\tilde{M}_1 \ T_{F,low} \tilde{M}_2$, and $\tilde{M}_1, \tilde{M}_2$ both have type $\theta \text{ref}_j$ for some $\theta$ and $j$ such that $j \notin F$ low, and $\tilde{N}_1 \ T_{F,low} \tilde{N}_2$. Clearly we have that $\tilde{M}_1 \in H_{F,low}$, so by induction hypothesis also $\tilde{M}_2 \in H_{F,low}$. We distinguish two sub-cases:

- $\tilde{N}_2 \in H_{F,low}$. Then, $\tilde{M}_2, \tilde{N}_2 \in H_{F,low}$ where $\tilde{M}_2$ has type $\theta \text{ref}_j$ for some $\theta$ and $j$ such that $j \notin F$ low. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,low}$.
- $\tilde{N}_2 \notin H_{F,low}$. Then $\tilde{M}_1 \dagger$, and by Lemma 3.4.17 also $\tilde{M}_2 \dagger$. Therefore, since $\tilde{M}_2$ has type $\theta \text{ref}_j$ for some $\theta$ and $j$ such that $j \notin F$ low, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,low}$.

**Clause 7.** Here we have $M_1 = (\text{flow} F' \text{ in } \tilde{M}_1)$ and $M_2 = (\text{flow} F' \text{ in } \tilde{M}_2)$ with $\tilde{M}_1 \ T_{F \cup F',low} \tilde{M}_2$. Clearly we have that $\tilde{M}_1 \in H_{F,low}$, so by induction hypothesis also $\tilde{M}_2 \in H_{F,low}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,low}$.

**Clause 8.** Here we have $M_1 = (\kappa \times \tilde{M}_1)$ and $M_2 = (\kappa \times \tilde{M}_2)$ with $\tilde{M}_1 \ T_{F,low} \tilde{M}_2$. Clearly we have that $\tilde{M}_1 \in H_{F,low}$, so by induction hypothesis also $\tilde{M}_2 \in H_{F,low}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,low}$.
Clause 8. Here we have $M_1 = (\text{enable } \kappa \text{ in } \tilde{M}_1)$ and $M_2 = (\text{enable } \kappa \text{ in } \tilde{M}_2)$ with $\tilde{M}_1 \ T_{F,\text{low}} \tilde{M}_2$. Clearly we have that $\tilde{M}_1 \in \mathcal{H}_{F,\text{low}}$, so by induction hypothesis also $\tilde{M}_2 \in \mathcal{H}_{F,\text{low}}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in \mathcal{H}_{F,\text{low}}$.

Behavior of Typable Low Expressions

In this second phase of the proof, we consider the general case of expressions that are typable with any termination level. As in the previous subsection, we show that a typable expression behaves as a strong bisimulation, provided that it is operationally low. For this purpose, we make use of the properties identified for the class of low-terminating expressions by allowing only these to be followed by low-writes. Conversely, high-terminating expressions can only be followed by high-expressions (see Definitions 3.3.4 and 3.4.9).

We now design a binary relation on expressions that uses $T_{F,\text{low}}$ to ensure that high-terminating expressions are always followed by operationally high ones. The definition of $R_{G,F,\text{low}}$, abbreviated $R_{F,\text{low}}$ when the global flow policy is $G$, is given in Figure 3.5. The flow policy $F$ is assumed to contain $G$. Notice that it is a symmetric relation. In order to ensure that expressions that are related by $R_{F,\text{low}}$ perform the same changes to the low memory, its definition requires that the references that are created or written using (potentially) different values are high, and that the body of the functions that are applied are syntactically high.

Remark 3.4.21 If $M_1 \ T_{F,\text{low}} M_2$, then $M_1 \ R_{F,\text{low}} M_2$.

The above remark is used to prove the following lemma.

Lemma 3.4.22 If for some $F$ we have that $M_1 \ R_{F,\text{low}} M_2$ and $M_1 \in \mathcal{H}_{F,\text{low}}$, then $M_2 \in \mathcal{H}_{F,\text{low}}$.

Proof sketch. Similarly to Lemma 3.4.19, the proof rests on the fact that if an expression $M_1$ of the form $(M_1 \tilde{N}_1), (M_1; \tilde{N}_1) \text{ or } (M_1 := \tilde{N}_1)$ is operationally high, in spite of $\tilde{N}_1$ not being operationally high, then $M_1$ is non-resolvable.

Clauses 5', 7' and 11' do not require $\tilde{N}_1$ to be operationally high. However, by the above observation and by Lemma 3.4.17 this implies that $M_2$ is non-resolvable. Therefore, it is sufficient to conclude that $\tilde{M}_2$ is operationally high.

Proof: By induction on the definition of $M_1 \ R_{F,\text{low}} M_2$.

Clause 1'. Direct.

Clause 2'. Direct.

Clause 3'. Here we have that $M_1 =$ if $\tilde{M}_1$ then $\tilde{M}_t$ else $\tilde{M}_f$ and that $M_2 =$ if $\tilde{M}_2$ then $\tilde{M}_t$ else $\tilde{M}_f$ with $\tilde{M}_1 \ R_{F,\text{low}} \tilde{M}_2$ and $\tilde{M}_t, \tilde{M}_f \in \mathcal{H}_{F,\text{low}}$. Clearly we have that $\tilde{M}_1 \in \mathcal{H}_{F,\text{low}}$, so by induction hypothesis also $\tilde{M}_2 \in \mathcal{H}_{F,\text{low}}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in \mathcal{H}_{F,\text{low}}$.

Clause 4'. Here $M_1 = (\tilde{M}_1 \tilde{N}_1)$ and $M_2 = (\tilde{M}_2 \tilde{N}_2)$ with $\tilde{M}_1 \ R_{F,\text{low}} \tilde{M}_2$, $\tilde{M}_1$ and $\tilde{M}_2$ are syntactically $(F,\text{low})$-high functions, and $\tilde{N}_1, \tilde{N}_2 \in \mathcal{H}_{F,\text{low}}$. Clearly we have that $\tilde{M}_1 \in \mathcal{H}_{F,\text{low}}$, so by induction hypothesis also $\tilde{M}_2 \in \mathcal{H}_{F,\text{low}}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in \mathcal{H}_{F,\text{low}}$. 
3.4 Type System Properties

Definition 3.4.20 \((R_{F,\text{low}})}\) We have that \(M_1 R_{F,\text{low}} M_2\) if \(\ell; F; \Gamma \vdash C M_1 : s_1, \tau\) and \(\ell; F; \Gamma \vdash C M_2 : s_2, \tau\) for some \(\ell, \Gamma, s_1, s_2\) and \(\tau\) and one of the following holds:

**Clause 1’.** \(M_1, M_2 \in H_{F,\text{low}},\) or

**Clause 2’.** \(M_1 = M_2,\) or

**Clause 3’.** \(M_1 = \text{if } M_1 \text{ then } \bar{N}_t \text{ else } \bar{N}_f \text{ and } M_2 = \text{if } M_2 \text{ then } \bar{N}_t \text{ else } \bar{N}_f \text{ with } M_1 R_{F,\text{low}} M_2,\) and \(\bar{N}_t, \bar{N}_f \in H_{F,\text{low}},\) or

**Clause 4’.** \(M_1 = (M_1 \bar{N}_1)\) and \(M_2 = (M_2 \bar{N}_2)\) with \(M_1 R_{F,\text{low}} M_2,\) and \(\bar{N}_1, \bar{N}_2 \in H_{F,\text{low}},\) and \(M_1, M_2\) are syntactically \((F,\text{low})\)-high functions, or

**Clause 5’.** \(M_1 = (M_1 \bar{N}_1)\) and \(M_2 = (M_2 \bar{N}_2)\) with \(M_1 T_{F,\text{low}} M_2,\) and \(\bar{N}_1, \bar{N}_2 \in H_{F,\text{low}},\) and \(M_1, M_2\) are syntactically \((F,\text{low})\)-high functions, or

**Clause 6’.** \(M_1 = (M_1 \bar{N})\) and \(M_2 = (M_2 \bar{N})\) with \(M_1 R_{F,\text{low}} M_2,\) and \(\bar{N} \in H_{F,\text{low}},\) or

**Clause 7’.** \(M_1 = (M_1 \bar{N})\) and \(M_2 = (M_2 \bar{N})\) with \(M_1 T_{F,\text{low}} M_2,\) or

**Clause 8’.** \(M_1 = (\text{ref}_i M_1)\) and \(M_2 = (\text{ref}_i M_2)\) with \(M_1 R_{F,\text{low}} M_2,\) and \(i \notin F\) low, or

**Clause 9’.** \(M_1 = (! M_1)\) and \(M_2 = (! M_2)\) with \(M_1 R_{F,\text{low}} M_2,\) or

**Clause 10’.** \(M_1 = (M_1 := \bar{N}_1)\) and \(M_2 = (M_2 := \bar{N}_2)\) with \(M_1 R_{F,\text{low}} M_2,\) and \(\bar{N}_1, \bar{N}_2 \in H_{F,\text{low}},\) and \(\bar{N}_1, \bar{N}_2 \in H_{F,\text{low}},\) and \(M_1, M_2\) both have type \(\theta\) \(\text{ref}_{j}\) for some \(\theta\) and \(j\) such that \(j \notin F\) low, or

**Clause 11’.** \(M_1 = (M_1 := \bar{N}_1)\) and \(M_2 = (M_2 := \bar{N}_2)\) with \(M_1 T_{F,\text{low}} M_2,\) and \(\bar{N}_1, \bar{N}_2 \in H_{F,\text{low}},\) and \(\bar{N}_1, \bar{N}_2 \in H_{F,\text{low}},\) and \(M_1, M_2\) both have type \(\theta\) \(\text{ref}_{j}\) for some \(\theta\) and \(j\) such that \(j \notin F\) low, or

**Clause 12’.** \(M_1 = (\text{flow } F' \text{ in } \bar{M}_1)\) and \(M_2 = (\text{flow } F' \text{ in } \bar{M}_2)\) with \(M_1 R_{F,F',\text{low}} \bar{M}_2,\)

**Clause 13’.** \(M_1 = (\tau \times M_1)\) and \(M_2 = (\tau \times M_2)\) with \(M_1 R_{F,\text{low}} M_2,\)

**Clause 14’.** \(M_1 = (\text{enable } i \text{ in } \bar{M}_1)\) and \(M_2 = (\text{enable } i \text{ in } \bar{M}_2)\) with \(M_1 R_{F,\text{low}} M_2,\)

---

Figure 3.5: The relation \(R_{F,\text{low}}\)

---

**Clause 5’.** Here \(M_1 = (M_1 \bar{N}_1)\) and \(M_2 = (M_2 \bar{N}_2)\) with \(M_1 T_{F,\text{low}} M_2,\) and \(M_1 \) and \(M_2\) are syntactically \((F,\text{low})\)-high functions, and \(\bar{N}_1 R_{F,\text{low}} \bar{N}_2.\) Clearly we have that \(M_1 \in H_{F,\text{low}},\) so by Lemma 3.4.19 also \(M_2 \in H_{F,\text{low}}.\) We distinguish two sub-cases:

\(\bar{N}_1 \in H_{F,\text{low}}.\) Then, by induction hypothesis, also \(\bar{N}_2 \in H_{F,\text{low}}.\) Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that \(M_2 \in H_{F,\text{low}}.\)

\(\bar{N}_1 \notin H_{F,\text{low}}.\) Then \(M_1 \dagger,\) and by Lemma 3.4.17 also \(M_2 \dagger.\) Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that \(M_2 \in H_{F,\text{low}}.\)

**Clause 6’.** Here \(M_1 = (M_1 \bar{N})\) and \(M_2 = (M_2 \bar{N})\) where \(M_1 R_{F,\text{low}} M_2\) and \(\bar{N} \in H_{F,\text{low}}.\) Clearly we have that \(M_1 \in H_{F,\text{low}},\) so by induction hypothesis also \(M_2 \in H_{F,\text{low}}.\) Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that \(M_2 \in H_{F,\text{low}}.\)
3.4 Type System Properties

Clause 7’. Here $M_1 = (N_1; \tilde{N})$ and $M_2 = (\tilde{M}_2; \tilde{N})$ with $\tilde{M}_2 \leftarrow F_{\text{low}} \tilde{M}_2$. Clearly we have that $\tilde{M}_1 \in H_{F,\text{low}}$, so by Lemma 3.4.19 also $M_2 \in H_{F,\text{low}}$. We distinguish two sub-cases:

\[ \tilde{N} \in H_{F,\text{low}}. \] Then, $\tilde{M}_2, \tilde{N} \in H_{F,\text{low}}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,\text{low}}$.

\[ \tilde{N} \notin H_{F,\text{low}}. \] Then $M_1 \uparrow$, and by Lemma 3.4.17 also $M_2 \uparrow$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,\text{low}}$.

Clause 8’. Here $M_1 = (\text{ref}_{j,\theta} \bar{M}_1)$ and $M_2 = (\text{ref}_{j,\theta} \bar{M}_2)$ where $\bar{M}_1 \rightarrow F_{\text{low}} \bar{M}_2$, and $j \nonl F$. Clearly we have that $\bar{M}_1 \in H_{F,\text{low}}$, so by induction hypothesis also $M_2 \in H_{F,\text{low}}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,\text{low}}$.

Clause 9’. Here $M_1 = (\text{!} \bar{M}_1)$ and $M_2 = (\text{!} \bar{M}_2)$ where $\bar{M}_1 \rightarrow F_{\text{low}} \bar{M}_2$. Clearly we have that $\bar{M}_1 \in H_{F,\text{low}}$, so by induction hypothesis also $M_2 \in H_{F,\text{low}}$. This implies that $M_2 \in H_{F,\text{low}}$.

Clause 10’. Here we have $M_1 = (\bar{M}_1 := N_1)$ and $M_2 = (\bar{M}_2 := N_2)$ where $\bar{M}_1 \rightarrow F_{\text{low}} \bar{M}_2$, and $\bar{M}_1, \bar{M}_2$ both have type $\text{ref}_j$ for some $\theta$ and $j$ such that $j \nonl F$. Clearly we have that $\bar{M}_1 \in H_{F,\text{low}}$, so by Lemma 3.4.19 also $M_2 \in H_{F,\text{low}}$. We distinguish two sub-cases:

\[ \bar{N} \in H_{F,\text{low}}. \] Then, $\tilde{M}_2, \tilde{N} \in H_{F,\text{low}}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,\text{low}}$.

\[ \bar{N} \notin H_{F,\text{low}}. \] Then $M_1 \uparrow$, and by Lemma 3.4.17 also $M_2 \uparrow$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,\text{low}}$.

Clause 12’. Here $M_1 = (\text{flow} F' \bar{M}_1)$ and $M_2 = (\text{flow} F' \bar{M}_2)$ with $\bar{M}_1 \rightarrow F_{\text{low}} \bar{M}_2$. Clearly we have that $\bar{M}_1 \in H_{F,\text{low}}$, so by induction hypothesis also $M_2 \in H_{F,\text{low}}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,\text{low}}$.

Clause 13’. Here $M_1 = (\text{!} \bar{M}_1)$ and $M_2 = (\text{!} \bar{M}_2)$ with $\bar{M}_1 \rightarrow F_{\text{low}} \bar{M}_2$. Clearly we have that $\bar{M}_1 \in H_{F,\text{low}}$, so by induction hypothesis also $M_2 \in H_{F,\text{low}}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,\text{low}}$.

Clause 14’. Here $M_1 = (\text{enable} \; \bar{M}_1)$ and $M_2 = (\text{enable} \; \bar{M}_2)$ with $\bar{M}_1 \rightarrow F_{\text{low}} \bar{M}_2$. Clearly we have that $\bar{M}_1 \in H_{F,\text{low}}$, so by induction hypothesis also $M_2 \in H_{F,\text{low}}$. Therefore, by Composition of High Expressions (Lemma 3.4.18) we have that $M_2 \in H_{F,\text{low}}$. 

We have seen in Splitting Computations (Lemma 3.1.2) that two computations of the same expression can split only if the expression is about to read a reference that is given different values by the memories in which they compute. Finally, from the following lemma one can conclude that the above relation $R_{F,\text{low}}$ relates the possible outcomes of typable expressions in general.

Lemma 3.4.23 If there exist $\ell, \Gamma, s, \tau$ such that $\ell; F; \Gamma \vdash_G \bar{E}[[\text{!} \alpha_{j,\theta}]] : s, \tau$ with $j \nonl_{F,\text{low}} \bar{E}$, then for any values $V_0, V_1 \in \text{Val}$ such that $\ell; F; \Gamma \vdash_G V_i : \theta$ we have $E[V_0] \rightarrow_{F,\text{low}} E[V_1]$. 


Proof sketch. If a typable expression is about to use a value that results from a high dereference, then the following situations can occur:
If the termination effect is low, i.e. if the value cannot influence the termination behavior of the dereference, then by Lemma 3.4.14 any two possible outcomes are in the relation $T$ (see Clauses 5', 7' and 11').
Otherwise, if the terminating effect is not low, then the type system must ensure that no low writes follow the high dereference (see Clauses 4', 6' and 10'). This is partly guaranteed by conditions of the form '$s.t \preceq_F s'.w$', where $s$ is the security effect of a subprogram that is performed before another subprogram whose security effect is $s'$. More precisely, conditions are imposed when the dereferenced value is used: to determine a reference that is being assigned to ($s.t \preceq_F s'.w$ in rule ASSIGN); to determine a function that is being applied ($s.t \preceq_F s'.w$ in rule APP); to evaluate the first component of a sequential composition ($s.t \preceq_F s'.w$ in rule SEQ).

The relation $R$ requires that the references that are created or written using the high dereferenced value are high (see Clauses 8', 10' and 11'), and that function applications that use the high dereferenced value are syntactically high. This is partly guaranteed by conditions of the form '$s.t \preceq_F j'$, where $s$ is the security effect of a subprogram that performs the high dereference, and $j$ is the security level of the reference that is created or written. More precisely, conditions are imposed when the dereferenced value is used: to create a reference ($s.r \preceq_F j$ in rule REFS); to determine a reference that is being assigned to ($s.r \preceq_F j$ in rule ASSIGN); to determine a value that is being assigned ($s'.r \preceq_F j$ in rule ASSIGN); to determine a function that is being applied ($s.r \preceq_F s'.w$ in rule APP); to determine an argument to which a function is being applied ($s'.r \preceq_F s'.w$ in rule APP).

When the high dereferenced value is used in the predicate of a conditional, the branches should be operationally high (see Clause 3'). This is guaranteed by the type system with the condition $s.r \preceq_F s_1.w, s_f.w$ in rule COND.

Proof: By induction on the structure of $E$.

$E[(! a_{j,θ})] = (! a_{j,θ})$. We have $V_0 \mid R_{F,low} V_1$ by Clause 1'.

$E[(! a_{j,θ})] = (E_1[(! a_{j,θ})] \cdot M)$. By rule APP we have $\ell; F; Γ \vdash_G E_1[(! a_{j,θ})] : \tilde{s}, \tilde{r} \rightarrow_{F,low} \tilde{s}' \cdot \tilde{σ}$ and $\ell; F; Γ \vdash_G M : \tilde{s}''.\tilde{r}$ with $\tilde{s}.r \preceq_F \tilde{s}'.w$ and $\tilde{s}.t \preceq_F \tilde{s}''.w$. By Lemma 3.4.1, we have $j \preceq_F \tilde{s}''.w$. Since by hypothesis $j \not\preceq_F low$ (therefore $\ell \not\preceq_F low$), then $\tilde{s}'.w \not\preceq_F low$, that is $E_1[(! a_{j,θ})]$ is a syntactically $(F,low)$-high function. By Lemma 3.4.5, the same holds for $E_1[V_0]$ and $E_1[V_1]$. By induction hypothesis we conclude that $E_1[V_0] \mid R_{F,low} E_1[V_1]$.

$\tilde{s}.t \preceq_F low$. Then $\tilde{s}''.w \not\preceq_F low$ (and also $\tilde{s}'.w \not\preceq_F low$) so by High Expressions (Lemma 3.4.10) we have $M \in H_{F,low}$. Therefore, we conclude $E[V_0] \mid R_{F,low} E[V_1]$ by Clause 4' and Lemma 3.4.5.

$\tilde{s}.t \preceq_F low$. Then by Lemma 3.4.14 we have $E_1[V_0] \mid T_{F,low} E_1[V_1]$. Therefore, since $M \mid R_{F,low} M$ by Clause 2', we conclude that $E[V_0] \mid R_{F,low} E[V_1]$ by Clause 5' and Lemma 3.4.5.

$E[(! a_{j,θ})] = (V E_1[(! a_{j,θ})])$. By rule APP we have that $\ell; F; Γ \vdash_G V : \tilde{s}, \tilde{r} \rightarrow_{F,low} \tilde{s}' \cdot \tilde{σ}$ and $\ell; F; Γ \vdash_G E_1[(! a_{j,θ})] : \tilde{s}''.\tilde{r}$ with $\tilde{s}'.r \preceq_F \tilde{s}''.w$. By Lemma 3.4.1, we have $j \preceq_F \tilde{s}''.r$, and so $j \preceq_F \tilde{s}''.w$. Since by hypothesis $j \not\preceq_F low$ (therefore $j \not\preceq_F low$), then $\tilde{s}'.w \not\preceq_F low$, that is $V$ is a syntactically $(F,low)$-high function. By Clause 1 we have $V \mid T_{F,low} V$. By induction hypothesis $E_1[V_0] \mid R_{F,low} E_1[V_1]$. Therefore we conclude that $E[V_0] \mid R_{F,low} E[V_1]$ by Clause 5' and Lemma 3.4.5.

$E[(! a_{j,θ})] = if E_1[(! a_{j,θ})] then M_f else M_f$. By COND we have that $\ell; F; Γ \vdash_G E_1[(! a_{j,θ})] : \tilde{s}, \tilde{r}$ and $\ell; F; Γ \vdash_G M_f : \tilde{s}_f, \tilde{r}$ with $\tilde{s}.r \preceq_F \tilde{s}_f.w, \tilde{s}_f.w$. By Lemma 3.4.1, we have $j \preceq_F \tilde{s}.r$ and so $j \preceq_F \tilde{s}_f.w$. Therefore, we conclude that $E[V_0] \mid R_{F,low} E[V_1]$ by Clause 5' and Lemma 3.4.5.
\[ \bar{s}.t \not\preceq_F \text{low} \] and \( \bar{s}.f.w \not\preceq_F \text{low} \). This implies that \( M_1, M_f \in \mathcal{H}_{F, \text{low}} \). By induction hypothesis \( E_1[V_0] \mathcal{R}_{F, \text{low}} E_1[V_1] \). Therefore, we conclude that \( E[V_0] \mathcal{R}_{F, \text{low}} E[V_1] \) by Clause 3' and Lemma 3.4.5.

\[ E[(! a_{j, \theta})] = (E_1[(! a_{j, \theta})] ; M) \] By \( \text{SEQ} \) we have \( \ell; F; \Gamma \vdash_G E_1[(! a_{j, \theta})] : \bar{s}, \bar{t} \) and \( \ell; F; \Gamma \vdash_G M : \bar{s}', \bar{t}' \) with \( \bar{s}.t \preceq_F \bar{s}'.w \).

\[ \bar{s}.t \not\preceq_F \text{low} \] Then \( \bar{s}'.w \not\preceq_F \text{low} \), so by \( \text{High Expressions (Lemma 3.4.10)} \) we have \( M \in \mathcal{H}_{F, \text{low}} \). By induction hypothesis \( E_1[V_0] \mathcal{R}_{F, \text{low}} E_1[V_1] \). We then conclude that \( E[V_0] \mathcal{R}_{F, \text{low}} E[V_1] \) by Clause 6' and Lemma 3.4.5.

\[ \bar{s}.t \not\preceq_F \text{low} \] Then by Lemma 3.4.14 we have \( E_1[V_0] \mathcal{T}_{F, \text{low}} E_1[V_1] \). Therefore, we conclude using Clause 7' and Lemma 3.4.5.

\[ E[(! a_{j, \theta})] = (\text{ref}_{\ell, \theta} E_1[(! a_{j, \theta})]) \] By \( \text{REF} \) we have \( \ell; F; \Gamma \vdash_G E_1[(! a_{j, \theta})] : \bar{s}, \bar{t} \) with \( \bar{s}.r = s.r \preceq_F \bar{\ell} \) and \( \bar{s}.t = s.t \). Therefore, since \( j \not\preceq_{F \cup \{E_1\}} \) implies \( j \not\preceq_{F \cup \{E_1\}} \), then by induction hypothesis we have \( E_1[V_0] \mathcal{R}_{F, \text{low}} E_1[V_1] \). By Lemma 3.4.1 we have \( j \not\preceq_F s.r \), so \( s.r \not\preceq_F \text{low} \). Therefore, \( \bar{\ell} \not\preceq_F \text{low} \), and we conclude by Lemma 3.4.5 and Clause 8'.

\[ E[(! a_{j, \theta})] = (E_1[(! a_{j, \theta})] \iff M) \] By rule \( \text{DEREF} \) we have \( \ell; F; \Gamma \vdash_G E_1[(! a_{j, \theta})] : \bar{s}, \bar{t} \). By induction hypothesis \( E_1[V_0] \mathcal{T}_{F, \text{low}} E_1[V_1] \). We then conclude by Lemma 3.4.5 and Clause 9'.

\[ E[(! a_{j, \theta})] = (E_1[(! a_{j, \theta})] \iff M) \] By rule \( \text{ASSIGN} \) we have that \( \ell; F; \Gamma \vdash_G E_1[!a_{j, \theta}] : \bar{s}, \bar{t} \) ref with \( \bar{s}.r \preceq_F \bar{\ell} \) and \( \bar{s}.t \preceq_F \bar{s}'.w \). By Lemma 3.4.1 we have \( j \not\preceq_F s.r \), so \( s.r \not\preceq_F \text{low} \) and so \( \bar{\ell} \not\preceq_F \text{low} \).

\[ \bar{s}.t \not\preceq_F \text{low} \] Then \( \bar{s}'.w \not\preceq_F \text{low} \), so by \( \text{High Expressions (Lemma 3.4.10)} \) we have \( M \in \mathcal{H}_{F, \text{low}} \). By induction hypothesis \( E_1[V_0] \mathcal{R}_{F, \text{low}} E_1[V_1] \). We then conclude that \( E[V_0] \mathcal{R}_{F, \text{low}} E[V_1] \) by Clause 10' and Lemma 3.4.5.

\[ \bar{s}.t \not\preceq_F \text{low} \] Then by Lemma 3.4.14 we have \( E_1[V_0] \mathcal{T}_{F, \text{low}} E_1[V_1] \). Therefore, we conclude using Lemma 3.4.5, Clause 11' and Clause 2' (regarding \( M \)).

\[ E[(! a_{j, \theta})] = (V \iff E_1[(! a_{j, \theta})]) \] By rule \( \text{FLOW} \) we have \( \ell; F \cup F'; \Gamma \vdash_G V : s, \tau \). By induction hypothesis \( E_1[V_0] \mathcal{T}_{F, \text{low}} E_1[V_1] \), so we conclude by Lemma 3.4.5 and Clause 12'.

\[ E[(! a_{j, \theta})] = (t \times E_1[(! a_{j, \theta})]) \] By rule \( \text{RESTRICT} \) we have \( \ell ; G ; t ; F ; \Gamma \vdash_G V : s, \tau \). By induction hypothesis \( E_1[V_0] \mathcal{T}_{F, \text{low}} E_1[V_1] \), so we conclude by Lemma 3.4.5 and Clause 13'.

\[ E[(! a_{j, \theta})] = (\text{enable } s \text{ in } E_1[(! a_{j, \theta})]) \] By rule \( \text{ENABLE} \) we have \( \ell ; G ; t ; F ; \Gamma \vdash_G V : s, \tau \). By induction hypothesis \( E_1[V_0] \mathcal{T}_{F, \text{low}} E_1[V_1] \), so we conclude by Lemma 3.4.5 and Clause 14'.

We now prove a crucial result of this chapter: the relation \( \mathcal{R}_{F, \text{low}} \) is a sort of ”strong bisimulation”.

**Theorem 3.4.24 (Strong Bisimulation for Typable Low Expressions)**

If \( M_1 \mathcal{R}_{F, \text{low}} M_2 \) and \( M_1 \not\in \mathcal{H}_{F, \text{low}} \) and \( \ell ; G ; M_1 ; \mu_1 \longrightarrow_{p'} (M_1', \mu_1') \), with \( \mu_1 \preceq_{F \cup \{p'\}, \text{low}} \mu_2 \) such that \( a \) is fresh for \( \mu_2 \) if \( a_{j, \theta} \in \text{dom} (\mu_1') - \text{dom} (\mu_1) \), then there exist \( M_2' \) and \( \mu_2' \) such that \( \ell ; G ; (M_2, \mu_2) \longrightarrow_{p'} (M_2', \mu_2') \) with \( M_1' \mathcal{R}_{F, \text{low}} M_2' \) and \( \mu_1' \preceq_{F, \text{low}} \mu_2' \).
The relation $M \rightarrow_{G, low} N$. If $M \in H_{G, low}$, then $M' \in H_{G, low}$, and we have $M' \rightarrow_{G, low} N$. The matching transition for $N$ is $\ell \vdash_G (N, \nu) \rightarrow (N, \nu)$, since $\text{dom}(\mu') \cap \text{dom}(\nu) = \text{dom}(\mu) \cap \text{dom}(\nu)$.

- $M \rightarrow_{G, low} L$, and $L, N \in H_{G, low}$. By induction hypothesis, there exist $L'$, such that $\ell \vdash_G (L, \mu) \rightarrow (L', \mu')$, where $M' \rightarrow_{G, low} L'$, $\mu' \equiv_{G, low} \mu$, $L' \in H_{G, low}$, therefore, $M' \rightarrow_{G, low} L'$, and $L', N \in H_{G, low}$. From this we get $M' \rightarrow_{G, low} N$, and the matching transition for $N$ is $\ell \vdash_G (N, \nu) \rightarrow (N, \nu)$, since $\text{dom}(\mu') \cap \text{dom}(\nu) = \text{dom}(\mu) \cap \text{dom}(\nu)$. 

We now state the main result of this chapter:

**Theorem 3.4.27 (Soundness)** If $M$ is typable in the context of the access level $\ell$, a global flow policy $G$ and a local policy $F$, that is if for some $\Gamma, s$ and $\tau$ we have $\ell; F; \Gamma \vdash_G M : s, \tau$, then $M$ satisfies the non-disclosure policy with respect to $F \cup G$, that is $M \in ND(F \cup G)$, for all $\ell'$.

**Proof:** By Clause 2 of Definition 3.4.20, for all choice of security level $low$, we have $M \rightarrow_{G, low} M$. By rule a) of Definition 3.4.25, we have $M \rightarrow_{G, low} M$. The soundness result is an immediate consequence of Proposition 3.4.26.
Chapter 4

Security Types for Sessions and Pipelines

4.1 Service Oriented Calculi

The development of the applications distributed over the web (web services) has triggered interest in development of a new computational paradigm called Service Oriented Computing (SOC). Services are heterogeneous computational entities, independently developed, with a high level of autonomy and varying reliability. When dealing with services, one would have to take into account handling the possible denial of service, verification of adequacy of the service, or deal with an abandonment of an active interaction. The SOC tends to develop languages for description and modeling of services and their behaviour, that would have to take all of this into account, providing mechanisms for programming decisions and handling consequences.

Therefore, a language for SOC would be expected to provide means of programming safe and secure, possibly complex interactions between clients and services. In that respect, an interaction is defined as a unit of activity in a service-oriented application, which is essentially a conversation between a client and an instance of a service. A conversation consists of exchange of messages, and invocation of subsidiary services. Another issue that has to be taken into account is combining of services in order to perform a computation, or into a new, more complex service. Such process is called orchestration, consisting of organizing an synchronizing data flow between different activities.

As a result of a joint research within the project SENSORIA, the Service Centered Calculus (SCC) [8] has emerged as an interesting proposal. In [21], the authors have proposed the session as a language construct for communication based programming. The SCC takes advantage of the session paradigm to model the client – service interaction. In order to ensure the adequacy of service invocation, the bodies of the client and the service are required to be mutually complementary, i.e. to follow certain predefined protocols, described by session types.

The Calculus of Services with Pipelines and Sessions (CaSPiS) [9] is a dataflow–oriented successor of the Service Centered Calculus (SCC) [8], in which communications can either follow fixed protocols (session) or be disciplined into data flows (pipelines), using the pipeline constructor inspired by Orc [22].

4.1.1 Services and Security Issues

Besides adequacy fulfillment, when dealing with web services, assurance needs be given that private data will not become visible to unauthorized bodies. One of the approaches to provide such guarantees is to introduce into the language mechanisms for controlling access rights [39, 4, 38, 33, 19, 10].

Consider, for example, a service which keeps track of grades of a University course. The name of the service is UnivRec, and it offers two possibilities of record manipulation. A student (code guarded with level student) is allowed to access his grade for a given course by sending the service his file–number (the
35

4.2 The Language

StudentId) and the number of the course (the CourseId), to which the service responds by outputting his grade as a result of a function grade. Only a teacher (code guarded with level teacher) can be allowed to update the grades by calling a service update, which changes the grade records for the given student and course ID.

Example 4.1.1

\[
\text{UnivRec. (student)} \triangleright ((\text{check}).\text{(StudentId)}.(\text{CourseId}).(\text{grade}(\text{CourseId}))
\]

\[
+ (\text{update}).\text{(StudentId)}.(\text{CourseId}).(\text{grade})(\text{fail})) \triangleright
\]

\[
\text{teacher} \triangleright ((\text{check}).\text{(StudentId)}.(\text{CourseId}).(\text{grade}(\text{CourseId}))
\]

\[
+ (\text{update}).\text{(StudentId)}.(\text{CourseId}).(\text{grade})(\text{success}).\text{update}(\text{CourseId}).(\text{grade})
\]

Assuming that the privileges of a student are less than those of a teacher, this service should provide both the interface for a student and a teacher, rejecting student’s attempt to update the grades (and in this example, informing him of the failure), while when activated by the teacher, the update operation is allowed, and the update service is activated on teacher’s behalf.

4.2 The Language

4.2.1 Syntax

The present calculus is essentially the calculus of [12] enriched with security levels. To this aim we have considered session types decorated by security levels and designed a type system that guarantees both safe communications and data security. As usual [6, 15] we assume a lattice of security levels: we use \( \ell, \iota, \kappa \) to range over security levels and \( \sqsubseteq, \sqcup, \sqcap \) to denote partial order, join and meet, respectively.

The syntactical constructs are listed in Figure 4.1, where syntax occurring only at runtime appears shaded, in order of non-increasing precedence. More precisely the prefixes in lines 2-7 have the precedence over the constructors on lines 8-11 and the remaining constructors have decreasing precedence, but for the restrictions which have the same precedence.

Values (ranged over by \( u, v, w \)) can be either basic values, variables, function calls, services or session names. In well-typed processes only the first two kinds of values can be exchanged, while all kinds of values can appear in the conditions of pattern matching.

- the nil process 0, as usual, denotes inactivity;
- the input construct \( (x).P \), inherited from process calculi, such as \( \pi \)-calculus, receives a value \( x \) and continues as \( P \). Input construct binds the name of the value, which acts as a placeholder until replacement by the actual value when communication occurs;
- the output process \( \langle x \rangle P \), dual of input, emits the value \( x \) and then continues as \( P \);
- the label sum, or the label branching \( \sum_{i=1}^{n} (l_i).P_i \), offers multiple choice of continuation processes, guarded by different labels, for external choice;
- the label choice \( \langle l \rangle.P \), or internal choice, is a dual of label selection, and selects one of the labels from the label sum when the synchronization occurs;
- the return construct return \( v.P \) emits the value \( v \) to the parent session, and continues as \( P \);
 Processes $P, Q, R ::=$ nil
 | $(x).P$ input
 | $(v).P$ output
 | return $v.P$ value return
 | $(l).P$ label choice
 | $s.P$ service definition
 | $\pi.P$ service invocation
 | $\sum_{i=1}^{n} (l_i).P_i$ label-guarded sum
 | $\bigcup_{i=1}^{n} \ell_i \times P_i$ level-guarded sum
 | $\ell \triangleright P$ framed process
 | $\tau \triangleright P$ session
 | $P > x > Q$ pipe
 | $P | Q$ parallel composition
 | if $u = v$ then $P$ else $Q$ matching
 | $(\nu s)P$ service restriction
 | $(\nu r)P$ session restriction
 Values $u, v, w ::=$ base value
 | $x$ variable
 | $f(v)$ function call
 | $s$ service
 | $r$ session

 Polarities $p ::= +, -$
4.2 The Language

\[ \text{(INV)} \quad \mathcal{D}[\ell \downarrow \tau.P, s.Q] \rightarrow (\nu r) \mathcal{D}[r^- \rhd \ell \downarrow P, r^+ \rhd \ell \downarrow Q | s.Q] \]
\[ r \notin \text{fn}(\mathcal{D}[\tau.P, s.Q]) \]

\[ \text{(COM)} \quad \mathcal{D}_r[(x).P, (v).Q] \rightarrow \mathcal{D}_r[P[v/x], Q] \]

\[ \text{(LCOM)} \quad \mathcal{D}_r[\sum_{i=1}^{n} (l_i).P_i, (l_k).Q] \rightarrow \mathcal{D}_r[P_k, Q] \]

\[ \text{(RET)} \quad \mathcal{D}_r[(x).P, C_{rr}[\text{return } v.Q]] \rightarrow \mathcal{D}_r[P[v/x], C_{rr}[Q]] \]

\[ \text{(PIPE)} \quad \mathcal{C}[(v).P > x > Q] \rightarrow \mathcal{C}[P > x > Q | Q[v/x]] \]

\[ \text{(PIPERET)} \quad \mathcal{C}[C_{rr}[\text{return } v.P] > x > Q] \rightarrow \mathcal{C}[C_{rr}[P] > x > Q | Q[v/x]] \]

\[ \text{(IFT)} \quad \mathcal{C}[\text{if } u = v \text{ then } P \text{ else } Q] \rightarrow \mathcal{C}[P](u = v) \downarrow \text{true} \]

\[ \text{(IFF)} \quad \mathcal{C}[\text{if } u \neq v \text{ then } P \text{ else } Q] \rightarrow \mathcal{C}[Q](u \neq v) \downarrow \text{false} \]

\[ \text{(LEVSEL)} \quad \mathcal{C}[\ell \downarrow \bigcup_{i=1}^{m} \ell_i \times P_i] \rightarrow \mathcal{C}[\ell \downarrow P_i] \quad \text{if } \ell_i \subseteq \ell \]

\[ \text{(END)} \quad \mathcal{C}[\ell \downarrow 0] \rightarrow \mathcal{C}[0] \]

\[ \text{(SCOP)} \quad P \rightarrow P' \Rightarrow (\nu m) P \rightarrow (\nu m) P' \]

\[ \text{(STR)} \quad P \equiv P' \land P' \rightarrow Q' \land Q' \equiv Q \Rightarrow P \rightarrow Q \]

Figure 4.2: Reduction Rules

\[ P \mid 0 \equiv P \quad P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad (\nu m) 0 \equiv 0 \]

\[ (\nu m) (\nu m') P \equiv (\nu m') (\nu m) P \quad (P \mid Q) \equiv (\nu m) (P \mid Q) \quad \text{if } m \notin \text{fn}(Q) \]

\[ ((\nu m) P) > x > Q \equiv (\nu m) (P > x > Q) \quad \text{if } m \notin \text{fn}(Q) \]

\[ r^p \rhd (\nu m) P \equiv (\nu m) (r^p \rhd P) \quad \text{if } r \neq m \]

\[ \ell \downarrow ((\nu m) P) \equiv (\nu m) (\ell \downarrow P) \]

\[ (P \mid R) > x > Q \equiv (P > x > Q) \mid (R > x > Q) \quad \ell \downarrow (P \mid Q) \equiv \ell \downarrow P \mid \ell \downarrow Q \quad 0 > x > Q \equiv 0 \]

\[ (r^p \rhd 0) > x > Q \equiv r^p \rhd 0 \quad 0 \equiv r^p \rhd 0 \equiv r^p \rhd P \mid r^p \rhd 0 \]

Figure 4.3: Structural Equivalence

A session name has two polarized session ends, the client side \(-\) and the service side \(+\), one being dual to the other. At runtime fresh session names with opposite polarities are generated and restricted. All communications of values and labels are executed inside the scopes of the same session names with opposite polarities.

4.2.2 Operational Semantics

The operational semantics by means of reduction rules is given in Figure 4.2, as proposed in the appendix of the technical report of the original paper [12], instead of a labeled transition system as in the final version of [12]. We let \( m \) range over service and session names.

We use evaluation contexts with either one or two holes. The evaluation contexts with two holes are necessary in order to model the interaction between two processes. Moreover we express through contexts the condition that a process is in the scope of some session name with a given polarity. More precisely we define the following four kinds of evaluation contexts:

\[ \mathcal{C} \quad ::= \quad [\cdot] \mid \mathcal{C} \mid P \mid r^p \rhd \mathcal{C} \mid C \quad (\nu m) \mathcal{C} \quad ::= \quad r^p \rhd C \mid C \quad (\nu m) \mathcal{C} \quad ::= \quad \mathcal{C}[\ell \downarrow [\cdot]] \mid \mathcal{C}[\ell \downarrow [\cdot]] \]

\[ \mathcal{C}_{rr} \quad ::= \quad r^p \rhd \mathcal{C} \mid \mathcal{C} \quad (\nu m) \mathcal{C} \quad ::= \quad \mathcal{C}_r \mid \mathcal{C}_{rr} \quad (\nu m) \mathcal{C} \quad ::= \quad \mathcal{D}_r[\mathcal{C}_{rr}, \mathcal{C}_{rr}] \quad r \notin \text{fn}(\mathcal{D}) \]

The Figure 4.3 contains rules for the structural equivalence of processes. Rules in the first line are
standard rules for commutativity and associativity and neutrality of 0 w.r.t. parallel composition and garbage collection of restrictions for the exhausted processes. Rules in the second and the third line enable commutation of restrictions and scope extrusion w.r.t. parallel composition and pipe, session and security framing construct. The rules in the fourth line enable splitting the pipe and security framing between parallelly composed processes. The last but one line contains garbage collection rules for pipes of exhausted processes and sessions. The rules in the last line are, respectively, garbage collection of exhausted sessions and un-nesting of exhausted session. In rule (INV) a client of level ℓ calls a service: the body of the client and a copy of the body of the service are framed by the client level ℓ and prefixed by two occurrences of the freshly generated and private session name r with opposite polarities.

In rules (COM) and (LCOM) two processes in the scopes of the same session name with opposite polarities exchange values and labels, respectively. When a label is received, the execution of the label-guarded sum continues with the corresponding process.

Rule (RET) allows the inner session (named r) to communicate the value v to the outer session (named r1).

In rules (Pipe) and (Piperet) a new parallel copy of Q is created: in this copy the value v replaces the variable x. The difference between these rules is that in the first one the value v is sent by an output prefix, while in the second one the value v is sent by a return prefix that occurs in an inner session (named r).

The rule (LEVSEL) non deterministically chooses a process whose security level is less than or equal to the level of the current frame.

Finally, rule (END) discards the security framing of an exhausted process. It has not been formulated as a rule of structural equivalence in order to preserve the invariance of typability under structural equivalence of subjects.

The remaining rules are standard. The structural equivalence is as usual, but for the rules dealing with security levels and sessions. The last three rules are, respectively, garbage collection of pipes of exhausted code, garbage collection of exhausted sessions and pushing the exhausted nested session outside of the scope of the parent session. We also consider structurally equivalent processes which can be obtained by α-renaming bound and restricted variables.

It is interesting to notice that CaSPiS mixes linear communication inside and between sessions with non–linear communication due to the pipe construct.

We end this section with an example to illustrate process reduction.

Example 4.2.1 We can define a service sqr that computes the square of a given number by calling a service mul, which in turn returns the product of two given numbers. More precisely if we define the processes:

\[ C = \text{sqr}.(x).N \]
\[ N = \text{mul}.(x).\langle x \rangle. \langle x \rangle. \text{return} \ y. \ 0 \]
\[ M = \text{mul}.(x).\langle x \rangle. \langle x \times y \rangle. \ 0 \]
\[ P = \bot \langle \text{sqr}.(5). \langle y \rangle. \ 0 \]

then the reduction is as follows, omitting the restrictions:

1. \( C \mid M \vdash P \) (Inv)
2. \( r^− \perp \perp \langle x \rangle.0 \mid r^+ \perp \perp \langle x \rangle.0 \mid C \mid M \) (Com)
3. \( r^− \perp \perp \langle x \rangle.0 \mid r^+ \perp \perp \langle x \rangle.0 \mid C \mid M \) (Com)
4. \( r^− \perp \perp \langle x \rangle.0 \mid r^+ \perp \perp \langle x \rangle.0 \mid C \mid M \) (Com)
5. \( r^− \perp \perp \langle x \rangle.0 \mid r^+ \perp \perp \langle x \rangle.0 \mid C \mid M \) (Com)
6. \( r^− \perp \perp \langle x \rangle.0 \mid r^+ \perp \perp \langle x \rangle.0 \mid C \mid M \) (Com)
7. \( r^− \perp \perp \langle x \rangle.0 \mid r^+ \perp \perp \langle x \rangle.0 \mid C \mid M \) (Com)
8. \( r^− \perp \perp \langle x \rangle.0 \mid r^+ \perp \perp \langle x \rangle.0 \mid C \mid M \) (Com)
9. \( C \mid M \)
4.3 Type System

The type system is a security annotated variant of the type system in [12], consisting of sorts and session types.

Each session type $T$ describes a communication protocol, which may consist of:

- no communication, if $T = \text{end}$;
- the input of a value of type $S_\ell$ followed by the protocol described by $T'$, if $T = ?(S_\ell).T'$;

Figure 4.6: Subtyping Relation $\leq$

\[
\begin{align*}
\text{end} & = \text{end} \\
?_i(S_\ell).T & = ?_i(S_\ell).T' \\
!_i(S_\ell).T' & = ?_i(S_\ell).T' \\
\oplus\{l_1 : T_1, \ldots, l_n : T_n\} & = \&\{l_1 : T_1, \ldots, l_n : T_n\} \\
\&\{l_1 : T_1, \ldots, l_n : T_n\} & = \oplus\{l_1 : T_1, \ldots, l_n : T_n\} \\
\lor[l_1, \ldots, l_n] \propto T & = \lor[l_1, \ldots, l_n] \propto T' \\
\end{align*}
\]

Figure 4.5: Duality Mapping

\[
\begin{align*}
&\&\{l_1 : T_1, \ldots, l_n : T_n\} \leq \&\{l_i : T_i\}_{i \in I} \\
&\oplus\{l_i : T_i\}_{i \in I} \leq \oplus\{l_i : T_i\}_{i \in I} \\
&\lor[l_1, \ldots, l_n] \propto T \leq \lor[l_i] \propto T \text{ if } \exists j \in I. \ell_j = \bigcap_{\ell_i \in I} \ell_i \\
&\left(1 \leq i \leq n\right) \quad T_i \leq T_i' \Rightarrow \left\{ \begin{array}{l}
&\&\{l_1 : T_1, \ldots, l_n : T_n\} \leq \&\{l_i : T_i\}_{i \in I} \\
&\oplus\{l_i : T_i\}_{i \in I} \leq \oplus\{l_1 : T_1, \ldots, l_n : T_n\} \\
&\lor[l_1, \ldots, l_n] \propto T \leq \lor[l_i] \propto T \\
\end{array} \right.
\]

\[
\begin{align*}
&T \leq T' \Rightarrow \left\{ \begin{array}{l}
&?_i(S_\ell).T \leq ?_i(S_\ell).T' \\
&\lor[l_1, \ldots, l_n] \propto T \leq \lor[l_i] \propto T' \\
\end{array} \right.
\]

where $I \subseteq \{1, \ldots, n\}$. 

Figure 4.4: Syntax of Types
4.3 Type System

- τ⁻¹ ∈ B  (BasVal)

Γ, s : Sₜ ⊢ s : Sₜ (Service)  Γ, x : Sₜ ⊢ x : Sₜ (Var)

Γ ⊢ v⁽¹⁾ : S⁽¹⁾ₜ, ..., Γ ⊢ v⁽ⁿ⁾ : S⁽ⁿ⁾ₜ  τ⁻¹ ∈ B  ℓ₁ ∪ ... ∪ ℓₙ ⊆ ₁

Γ, f : S⁽¹⁾ₜ × ... × S⁽ⁿ⁾ₜ → τ⁻¹ ⊢ f(v⁽¹⁾, ..., v⁽ⁿ⁾) : τ⁻¹  (Fun)

Γ ⊢ 0 : end : ⊥ (Nil)

Γ ⊢ P : U : b{Tₜ}  Γ ⊢ v : [Tₜ]ₜ  T'' ≤ T

Γ ⊢ P : U : b{Tₜ}  Γ ⊢ v : Sₜ

Γ ⊢ return v.P !(Sₜ).U : b{Tₜ} (Out)

Γ ⊢ P : U : b{Tₜ} (Ret)

Γ ⊢ P_i : U : b{T_i} (Branch)

Γ ⊢ \sum_{i=1}^{n}(l_i).P_i : U : b{[l_1 : T_1, ..., l_n : T_n]}_{\ell_1 ∪ ... ∪ \ell_n}

Γ ⊢ (l_i).P : U : b{[l_1 : T_1, ..., l_n : T_n]}_{\ell_1 ∪ ... ∪ \ell_n} (Choice)

Γ ⊢ P : U : b{T_i}  Γ, x : Sₜ ⊢ Q : U'' : b{T''} (pipe(U : b{T_i}, U'' : b{T''})ₜ, Sₜ) = U'' : b{T''} (Pipe)

Γ ⊢ P > x > Q : U'' : b{T''} (If)

Γ ⊢ v_i : Sₜ  Γ ⊢ P : U : b{T_j} (If)

Γ ⊢ v_j : Sₜ ⊢ Q : U : b{T_j} (If)

Γ ⊢ P_i : U : b{T_i} ⨯ ℓ_i ⊆ ℓ_i  (1 ≤ i ≤ n)  Γ ⊢ P_j : U : b{T_i} ⨯ ℓ_i ⊆ ℓ_j (LevSel)

Γ ⊢ \mathbf{b}^{n}_{i=1} ℓ_i ⨯ P_i : U : b{[ℓ_1, ..., ℓ_n] ⨯ T_j} (LevSel)

Γ ⊢ P : U : b{T₁}ₜ  Γ ⊢ Q : U' : b{T₂}ₜ (Par)

Γ ⊢ P | Q : U o U' : b{T₁}ₜ ⨯ b{T₂}ₜ (Par)

Figure 4.7: Typing Rules for Inner Processes
4.3 Type System

pipe((U : : +[end]_κ, U' : : +[T']_κ, S) = U : : +[end]_κ
pipe(U : : +[S(S)]_κ, U' : : +[T']_κ, S) = U ◦ U' : : +[T']_κ ⊑ [S]_κ
pipe(U : : +[S(S)h]_κ, U' : : +[end]_κ, S) = U ◦ U' : : +[end]_κ ⊑ [S]_κ

Figure 4.8: Function pipe

Γ ⊢ 0 : : +[end]⊥ (Nil)
Γ ⊢ P : : +[T]_κ  Γ ⊢ s : [T]_κ ⊑ [ℓ]
Γ ⊢ (s) s.P : : +[end]_κ (ServDef)

Γ ⊢ S.1 P : : +[end]_κ  Γ ⊢ S.2 Q : : +[end]_κ
Γ ⊢ S.1 S.2 P | Q : : +[end]_κ (TopPar)

Γ ⊢ P : : +[end]_κ  Γ ⊢ [ℓ] ◁ P : : +[end]_κ (LevSign)

Γ, s : S ⊢ S.1 P : : +[end]_κ
Γ ⊢ (ν s) P : : +[end]_κ (ServRest)

Figure 4.9: Typing Rules for Top Level Processes

Γ ⊢ P : : +[end]_κ  Γ ⊢ U : : +[T]_κ  μ ⊑ [ℓ]  Γ ⊢ P : : +[T]_κ (LevSignRT)

Γ ⊢ P : : +[T]_κ  Γ, r : [T]_κ ⊢ P : : +[T]_κ (Sess)
Γ ⊢ P : : +[T]_κ  Γ, r : [T]_κ ⊢ r ⊑ P : : +[U]_κ (Sess)

Γ, r : S ⊢ P : : +[T]_κ  (SessRest)
Γ, r : S ⊢ (ν r) P : : +[T]_κ (SessRestTop)

Figure 4.10: Typing Rules for Runtime Processes
4.3 Type System

- the output of a value of type $S_i$ followed by the protocol described by $T'$, if $T = !(S_i).T'$;
- the reception of a label $l_i$ chosen in the set $\{l_1, \ldots, l_n\}$ followed by the protocol described by $T_i$, if $T = \&\{l_1 : T_1, \ldots, l_n : T_n\}$;
- the transmission of a label $l_i$ chosen in the set $\{l_1, \ldots, l_n\}$ followed by the protocol described by $T_i$, if $T = @\{l_1 : T_1, \ldots, l_n : T_n\}$;
- the choice of a process with security level $\ell_i \in \{\ell_1, \ldots, \ell_n\}$, with $\ell_i$ less than or equal to the current security level, followed by the protocol described by $T'$, if $T = \triangledown\{\ell_1, \ldots, \ell_n\} \times T'$.

Basic data types are decorated with their security levels (cf. [41]), and the security levels appearing in annotation of session types represent the join of the levels of data manipulated inside the session by communication and activation.

A typing environment $\Gamma$ is a finite partial mapping from variables, services and sessions to sorts (i.e. session types and basic types) and function types. The empty environment is denoted by $\emptyset$, and if $u$ is a variable, a service or a session name, then $\Gamma, u : S_i$ is the extension with the binding of $u$ to $S_i$, if $u \not\in \text{dom}(\Gamma)$. The extension for a function name with a function type is defined similarly.

The typing judgments have the form $\Gamma \vdash v : S_i$ for the values, and $\Gamma \vdash P : U : \ldots : T_\kappa$ for processes, where $U$ is the output (sequence of outputs) of the process $P$ to the parent session with the level $i$, and $T$ is the type of $P$'s activities in the current session, and $\kappa$ is the level of values communicated in the current session and in all of the nested sessions.

Rules (Service), (Var), (BasVal) and (Fun) are standard typing rules for values. Rule (Fun) inserts an external function in the typing environment while ensuring that the level of the function output is not less than the level of its arguments.

The type $\text{end} : \ldots \text{end}_\downarrow$ of $\emptyset$ (rule (Nil)) means that no action is performed neither in the current nor in the parent session.

Rules for input and output of a value ((Inp), (Out)) add the performed action to the current session type, and increase the level assigned to the current session by the level of the communicated value.

Rule (Ret) adds the output of a value to the return sequence $U$, updating its level in a similar way.

Rules (Branch) and (Choice) for internal and external choice take the join of levels of all the branches as the current session level.

Rule (If) is the standard conditional branching on the result of the comparison of two values, which does not allow low–level branches to depend on high–level values in the choice.

Rule (Par) is used to type the parallel composition of two processes. The composition is possible if at least one of them does not have any action in the current session. Therefore, the operator $\bowtie$ is defined only when one of the types is end:

$$T \bowtie T' = \begin{cases} T', & \text{if } T = \text{end} \\ T, & \text{if } T' = \text{end} \\ \text{undefined otherwise} \end{cases}$$

Two parallel processes can both offer outputs to the parent session when there is no way of distinguishing the outputs at the type system level. The composition of outputs used in rule rule (Par) is defined as $U \circ U' = !(S_j)^{h+k}.\text{end}$ if $U = !(S_j)^h.\text{end}$ and $U' = !(S_j)^k.\text{end}$, i.e. $U$ and $U'$ are sequences of the same output, and undefined otherwise.

Rule (Pipe) derives the type of pipe using the function pipe, given in Figure 4.8. If $P$ outputs a single value (and has no other action in the current session), the pipe $Q$ consumes the value, and may have other actions in the current session. If the current session activity of $P$ consists of a sequence of $h$ outputs where $h > 1$, $h$ instances of $Q$ will be activated to consume the outputs and put in parallel with $P$, but $Q$ will only be allowed to produce values upwards to the parent session. If $Q$ will be activated, the outputs of $P$ and $Q$ will be composed as in parallel composition.
The innovation with respect to the type system in [12] is rule (\text{LevSel}). It is similar to the test–rule in access control typing systems [33, 19, 10]. Each branch of the choice is guarded with a security level greater than or equal to the one necessary for its activation. The level of the session type is increased to take into account only the least of the branch–levels. Since all the branches have the same type, differing only in the security annotation, the process is required to have at least the privilege to activate the lowest one, but depending on the process privileges, one of the higher branches may be selected. This rule permits a better modularity of service definitions, allowing the same service to offer both high and low–level response, depending on the clearance of the client who activated it.

Rule (\text{Inv}) for session invocation allows a process to invoke a session if its body matches the defined session w.r.t the duality mapping, defined in Figure 4.5. The type of the defined session is required to be a subtype of the client’s dual (in the sense of the relation inspired by [20], defined in Figure 4.6). This is justified by reasoning that the service can offer more choices than required by the client (with label and level branching), and that different clients may make different choices. After the process has been prefixed with service invocation, the return sequence $U$ becomes the current session usage, and its level, together with the level of the invoked session is added to the level of the current session.

A top level process is a process of type $\text{end} : \bot$, which means that it has no activity in the current session, and produces no value upwards. The typing of the top level processes is given in Figure 4.9. The typing judgments of the top level processes have the form $\Gamma \vdash S P : \text{end} : \bot$, where $S$ is the set of defined services.

An initial configuration consists of parallel compositions of service definitions and framed top level client processes.

Rule (\text{ServDef}) defines a service with as body the process $P$. Each time a service is defined, the turnstile is decorated with its name, and when processes are parallelly composed (rule (\text{TopPar})), the sets of defined services are joined. When we want to restrict a service name (rule (\text{ServRestr})), it is required that this name is already defined (i.e. present in $S$), and after restricting it, we remove it from the set $S$. In this way we prevent calls of nonexistent services.

Rule (\text{LevSign}) promotes an inner process to the top level process $\ell \triangleleft P$ by assignment of a security clearance $\ell$ only if the intended activity of $P$ (reflected in the level $\kappa$) agrees with $\ell$.

By requiring that the initial configuration consists only of top level processes, we avoid stuck configurations of dangling communications outside of sessions waiting for synchronization.

The typing of the processes that appear only at runtime is given in Figure 4.10.

Rule (\text{ToTop}) promotes an inner process to become a top level process. Since this rule is only applied at runtime it can avoid to check the absence of dangling communications.

At runtime also inner processes can be framed, ad therefore rule (\text{LevSignRT}) is needed in order to type them.

Rules (\text{Sess}) and (\text{Sessl}), similarly to the rules for service invocation and definition, require a session matching the type of the session body process $P$ in the typing environment. The difference between these pairs of rules is that the assumptions for sessions are added in the conclusions of the typing rules, while the assumptions for the services must be always present in the premises of the typing rules. The reason is that we want to allow nested calls of the same service name, but not nested usages of the same session name.

Rules (\text{SessRestr}) and (\text{SessRestrTop}) restrict the session name and remove it from the environment. Two rules are needed since both inner and top level processes can be restricted.

**Example 4.3.1** We put your type assignment system at work to type some of the processes shown in Example 4.2.1, assuming the level of all integers and services being $\bot$.

Let $T = [?(\text{int}_\bot).?(\text{int}_\bot).!(\text{int}_\bot).\text{end}]_\bot$, $T' = [?(\text{int}_\bot).!(\text{int}_\bot).\text{end}]_\bot$, and $\Gamma = \{\text{mul} : [T]_\bot, \text{sqr} : [T']_\bot\}$. 
We can type $N$, $C$, $M$ and $P$ as follows:

\[
\begin{align*}
\Gamma, x : \texttt{Int}_\bot & \vdash 0 : \texttt{end} : \bot \{\text{Inv}\} \\
\Gamma, x : \texttt{Int}_\bot & \vdash \text{return } y. 0 : !\{\texttt{Int}_\bot\}. \text{end} : \bot \{\text{Out}\} \\
\Gamma, x : \texttt{Int}_\bot & \vdash x : \texttt{Int}_\bot \quad (\text{Rest}) \\
\Gamma, x : \texttt{Int}_\bot & \vdash \text{return } y. 0 : !\{\texttt{Int}_\bot\}. \text{end} : \bot \{\text{Out}\} \\
\Gamma, x : \texttt{Int}_\bot & \vdash x : \texttt{Int}_\bot \quad (\text{Out}) \\
\Gamma, x : \texttt{Int}_\bot & \vdash x : \texttt{Int}_\bot \quad (\text{Out}) \\
\Gamma, x : \texttt{Int}_\bot & \vdash x : \texttt{Int}_\bot \quad (\text{ServDef}) \\
\end{align*}
\]

By applying the rules for service restriction and parallel composition, we can conclude:

\[
\emptyset \vdash_0 (\nu \texttt{mul}) (\nu \texttt{sqr}) \langle C \mid M \mid P \rangle : \texttt{end} : \bot \{\text{Inv}\}
\]

We give now the typing for the process in line (3) of Example 4.2.1. We define $N^* = \texttt{mul}.(5).\langle x \mid y \rangle$. return $y. 0$: it is easy to verify that we can derive $\Gamma \vdash N^* : \texttt{end} : \bot \{\texttt{Int}_\bot\}. \text{end} : \bot$ with a derivation similar to that one for $N$. 
4.4 Example

Note that the types of the processes constituting the two bodies of the same session name with opposite polarities are each other’s dual.

With applications of rules (ToTop), (TopPar), and (ServRestr) the type of the entire configuration is:

\[ \emptyset \vdash \emptyset (\nu \, \text{mul}) (\nu \, \text{sqr}) ((\nu \, r) \, r^- \triangleright \ell \triangleright N^* \mid r^+ \triangleright \ell \triangleright (x).0) \mid M \mid C \colon \text{end} \colon \vdash \text{[end]} \]

We end the example inferring types for some interesting subprocesses of the process in line (4) of Example 4.2.1.

\[ \begin{array}{lll}
\Gamma \vdash N^* \colon \text{end} & \vdash \text{[!([Int\downarrow]) \, \text{end}]} \\
\Gamma \vdash \perp \colon \text{N}^* \colon \text{end} & \vdash \text{[!([Int\downarrow]) \, \text{end}]} & \text{(LevSignRT)} \\
\Gamma, r : [T'] \vdash r^- \triangleright \ell \triangleright N^* \colon \text{end} & \vdash \text{[!([Int\downarrow]) \, \text{end}]} & \text{(Sess)} \\
\Gamma, r : [T'] \vdash r^+ \triangleright \ell \triangleright (x).0 \colon \text{end} & \vdash \text{[!([Int\downarrow]) \, \text{end}]} & \text{(Par)} \\
\end{array} \]

4.4 Example

We give an example of an e-commerce service, such as eBay. Three levels of users are assumed—guests, with level 1, which do not have an account with the service, and who are allowed only to browse through the items in the catalogue. The registered users have a level 2, and they are additionally allowed to bid for an item, or buy an item that is for direct sale, but are allowed to pay for it only by sending a personal cheque. The VIP–users with the level 3 can also pay with credit card or PayPal account. We assume that 1 \sqsubset 2 \sqsubseteq 3.
Example 4.4.1

\[ P = \langle \text{catalogue}\rangle \langle \text{itemNo}1\rangle \langle \text{price}\rangle \langle \text{itemNo}1\rangle \quad Q = \langle \text{pay}\rangle \langle \text{bid}1\rangle \langle \text{cheque}\rangle \langle \text{personalId}1\rangle, \]
\[ R = \langle \text{pay}\rangle \langle \text{bid}1\rangle \text{if } (\text{payMeth}_2(\text{personalId}1) = 1) \text{ then } \langle \text{cheque}\rangle \langle \text{personalId}1\rangle \]
else if (payMeth_3(\text{personalId}1) = 2) then \langle \text{card}\rangle \langle \text{cardNo}3(\text{PersonalId}2)\rangle \text{ else } \langle \text{paypal}\rangle \langle \text{accPars}_3(\text{PersonalId}2)\rangle \]
we can define the services ebay, pay, card and paypal as follows:

\[
\text{ebay}_1. \quad (1 \times ((\text{browse}),(\text{bid}),(\text{itemNo}1),(\text{bid}1),(\text{personalId}1))
+ (\text{buy}),(\text{itemNo}1),(\text{personalId}1))
\]
\[
\text{ebay}_2. \quad (2 \times ((\text{cheque}),(\text{personalId}1))
+ (\text{card}),(\text{cardPar}_2) + (\text{paypal}),(\text{accountPar}_2))
\]
\[
\text{ebay}_3. \quad (3 \times ((\text{cheque}),(\text{personalId}1))
+ (\text{card}),(\text{cardNo}_3),(\text{card}),(\text{amount}_1),(\text{cardNo}_2)
+ (\text{paypal}),(\text{accountPar}_2),
\text{paypal}),(\text{amount}_1),(\text{accountPar}_2))
\]
\[
\text{card}_3,(\text{amount}_1),(\text{cardNo}_2)
\]
\[
\text{paypal}_3,(\text{amount}_1),(\text{accountPar}_2)
\]

The service ebay offers essentially three kinds of services: browsing through the item catalogue, bidding for an item (with obligatory purchase of the customer who won the bid) and buying an item that is put for direct sale. Options for bidding and buying are different for users and VIP–users, since the VIP–users are allowed different payment methods based on their preference (retrieved by the function payMeth), while the ordinary users automatically have to pay by sending a personal cheque.

The payment is done by activating a service pay, which for a received amount triggers a payment method branch, which, in turn, might activate services that handle card and PayPal payments (services card and paypal, respectively).

The more interesting assumptions in the environment for typing ebay are (we omit the end terminators):

\[
\text{paypal} : [?((\text{int}1),(\text{string}2))_3], \text{card} : [?((\text{int}1),(\text{int}2))_3],
\text{pay} : [?((\text{int}1),(\text{int}2))_3] \times \{\text{cheque} : [?((\text{string}1),(\text{card}) : [?((\text{int}1),(\text{string}2))_2],
\text{ebay} : [?((\text{int}1),(\text{int}2))_3] \times \{((\text{browse}) : [?((\text{string}1),(\text{int}1))_2],
\text{bid} : [?((\text{int}1),(\text{int}2))_3],
\text{buy} : [?((\text{int}1),(\text{string}1))_2],
\text{accPars}_3(\text{PersonalId}2)\rangle \]
\]

Notice that although some options reserved for high users are formally (in name) present among the lower level choices, the actual activation of high–level services and communication of high–level data is reserved for high users.

4.5 Properties

4.5.1 Subject Reduction

Firstly, we will discuss the Subject Reduction property, namely that typability of expressions is preserved by the reduction relation. We start by showing standard auxiliary properties, such as strengthening, weakening,
typing invariance under structural equivalence and substitution.

**Lemma 4.5.1**

1. If $\Gamma, x : S_\ell \vdash P : U : \cdot [T]_\kappa$ and $x \notin \text{fv}(P)$, then $\Gamma \vdash P : U : \cdot [T]_\kappa$

2. If $\Gamma \vdash P : U : \cdot [T]_\kappa$, and $x \notin \text{dom}(\Gamma)$ then $\Gamma, x : S_\ell \vdash P : U : \cdot [T]_\kappa$.

**Lemma 4.5.2** (Invariance of Typing under Structural Equivalence) If $\Gamma \vdash P : U : \cdot [T]_\kappa$ and $P \equiv Q$ then $\Gamma \vdash Q : U : \cdot [T]_\kappa$.

**Proof:** by cases on the definition of structural equivalence. We give proofs only for interesting cases.

**Case** $(\nu r)$ $(r^+ \triangleright 0 \mid r^- \triangleright 0) \equiv 0$

The typing of the l.h.s. is:

$$
\Gamma \vdash 0 : \text{end} : \cdot [\text{end}]_\perp
$$

(Sess)

and the typing of the r.h.s. is:

$$
\Gamma \vdash 0 : \text{end} : \cdot [\text{end}]_\perp
$$

(Nil)

**Case** $r^p \triangleright (P \mid r^q_1 \triangleright 0) \equiv r^p \triangleright P \mid r^q_1 \triangleright 0$ We assume $p = q = +$, the other cases being similar. Then the typing of the l.h.s. is:

$$
\Gamma, r_1 : [\text{end}]_\perp \vdash P : U : \cdot [T]_\kappa
$$

(Sess)

and typing of the r.h.s. is:

$$
\Gamma, r_1 : [\text{end}]_\perp, r : [T]_\kappa \vdash r^+ \triangleright (P \mid r^+_1 \triangleright 0) : \cdot [U]_{\kappa \perp}.
$$

(Sess)

Figure 4.11: Consumption Relation $\vdash$
4.5 Properties

where we used weakening to add premises concerning \( r \) and \( r_1 \).

**Case \( \ell \upharpoonright (P \mid Q) \equiv \ell \upharpoonright P \mid \ell \upharpoonright Q \)**

Firstly, we consider the case in which the l.h.s is typed using rule (LevSign), where \( P \mid Q \) need to be of type end \( \vdash \) for some \( \kappa \subseteq \ell \).

\[
\frac{\Gamma \vdash P : \text{end} \vdash \kappa_1 \quad \Gamma \vdash Q : \text{end} \vdash \kappa_2}{\Gamma \vdash P \mid Q : \text{end} \vdash \kappa_1 \cup \kappa_2 \subseteq \ell} \quad \text{(Par)} \quad \kappa_1 \cup \kappa_2 \subseteq \ell
\]

The typing of the r.h.s. is as follows:

\[
\frac{\Gamma \vdash P : \text{end} \vdash \kappa_1 \quad \kappa_1 \subseteq \ell}{\Gamma \vdash \ell \upharpoonright P : \text{end} \vdash \kappa_1} \quad \text{(LevSign)} \quad \frac{\Gamma \vdash Q : \text{end} \vdash \kappa_2 \quad \kappa_2 \subseteq \ell}{\Gamma \vdash \ell \upharpoonright Q : \text{end} \vdash \kappa_2} \quad \text{(LevSign)}
\]

Secondly, we consider the case in which the l.h.s is typed using rule (LevSignRT), where \( P \) and \( Q \) can be of arbitrary types, which agree with the security level \( \ell \):

\[
\frac{\Gamma \vdash P : U :^{i_1}[T_1]\kappa_1 \quad \Gamma \vdash Q : U' :^{i_2}[T_2]\kappa_2}{\Gamma \vdash P \mid Q : U \circ U' :^{i_1[i_2]}[T_1 \diamond T_2]\kappa_1 \cup \kappa_2 \subseteq \ell} \quad \text{(Par)} \quad \kappa_1 \cup \kappa_2 \subseteq \ell
\]

The typing of the r.h.s. is as follows:

\[
\frac{\Gamma \vdash \ell \upharpoonright (P \mid Q) : U \circ U' :^{i_1[i_2]}[T_1 \diamond T_2]\ell}{\Gamma \vdash \ell \upharpoonright P : \ell \upharpoonright Q : U \circ U' :^{i_1[i_2]}[T_1 \diamond T_2]\ell} \quad \text{(LevSignRT)}
\]

\[
\frac{\Gamma \vdash \ell \upharpoonright P : \ell \upharpoonright Q : U \circ U' :^{i_1[i_2]}[T_1 \diamond T_2]\ell}{\Gamma \vdash \ell \upharpoonright P : U :^{i_1}[T_1]\kappa_1 \quad \kappa_1 \subseteq \ell \quad \kappa_1 \cup \kappa_2 \subseteq \ell \quad \Gamma \vdash \ell \upharpoonright Q : U' :^{i_2}[T_2]\kappa_2 \quad \kappa_2 \subseteq \ell \quad \Gamma \vdash \ell \upharpoonright Q : U' :^{i_2}[T_2]\kappa_2 \quad \kappa_2 \subseteq \ell} \quad \text{(LevSignRT)}
\]

\[
\frac{\Gamma \vdash \ell \upharpoonright P : \ell \upharpoonright Q : U \circ U' :^{i_1[i_2]}[T_1 \diamond T_2]\ell}{\Gamma \vdash \ell \upharpoonright P : U :^{i_1}[T_1]\kappa_1 \quad \kappa_1 \subseteq \ell \quad \kappa_1 \cup \kappa_2 \subseteq \ell \quad \Gamma \vdash \ell \upharpoonright Q : U' :^{i_2}[T_2]\kappa_2 \quad \kappa_2 \subseteq \ell \quad \Gamma \vdash \ell \upharpoonright Q : U' :^{i_2}[T_2]\kappa_2 \quad \kappa_2 \subseteq \ell} \quad \text{(Par)}
\]

\[
\begin{array}{l}
\frac{\Gamma \vdash P : U \vdash \kappa \quad \Gamma \vdash v : S_T}{\Gamma \vdash P[v/\ell] : U \vdash \kappa} \quad \text{Lemma 4.5.3 (Substitution)} \quad \text{Let} \quad \Gamma, x : S_T \vdash P : U :^{i}[T]_\kappa \quad \text{and} \quad \Gamma \vdash v : S_T. \quad \text{Then} \quad \Gamma \vdash P[v/\ell] : U :^{i}[T]_{\ell \cup \kappa}. \\
\end{array}
\]

In Figure 4.11 we introduce the relation \( < \) in order to express the consumption of the session types by reduction.

In the following lemma, we show that if a complex expression of C-context form is typable, then the subexpression inside the hole is also typable. Furthermore, we establish that a subexpression inside a C-context can be replaced with another subexpression of the same type with lower effects, while preserving the type of the entire expression, with a decrement of its effects.

**Lemma 4.5.4 (Replacement \( \supset \))** If \( \Gamma \vdash C[P] : U :^{i}[T]_\kappa \), then there exist some \( U_1, i_1, T_1, \kappa_1 \) such that \( \Gamma \vdash P : U_1 :^{i_1}[T_1]_{\kappa_1} \), and if there exists \( Q \) such that \( \Gamma \vdash Q : U_1 :^{i_2}[T_1]_{\kappa_2} \), \( i_2 \subseteq i_1 \), \( \kappa_2 \subseteq \kappa_1 \), then \( \Gamma \vdash C[Q] : U :^{i'}[T]_{\kappa'} \) for some \( i', \kappa' \) such that \( i' \subseteq i \), \( \kappa' \subseteq \kappa \).
Proof: by induction on the structure of $\mathbb{C}$ and by compositionality of the type system.

We will now show that if a complex expression of $\mathbb{D}$-context form is typable, then the two subexpressions inside the holes are also typable. If the two subexpressions in the holes are replaced with expressions of the same types and with lower joint effects, the typability of the entire expression will be preserved, with decrement of its effects.

Lemma 4.5.5 (Replacement $\mathbb{D}$) If $\Gamma \vdash \mathbb{D}[P, Q] : U : :^i[T]_\kappa$, then there exist some $U_1, T_1, \kappa_1$ and $U_2, T_2, \kappa_2$ such that $\Gamma \vdash P : U_1 : :^1[T]_\kappa$, and $\Gamma \vdash Q : U_2 : :^2[T]_\kappa$ and if there exist $P', Q'$ such that $\Gamma \vdash P' : U_1 : :^3[T]_\kappa$, and $\Gamma \vdash Q' : U_2 : :^4[T]_\kappa$, then $\Gamma \vdash \mathbb{D}[P', Q'] : U : :^i[T]_\kappa'$, for some $i, \kappa'$ such that $i' \subseteq i$, $\kappa' \subseteq \kappa$.

Proof: by induction on the structure of $\mathbb{D}$ and by compositionality of the type system.

Finally, we show that if a complex expression of $\mathbb{D}_r$-context form is typable, then the two subexpressions inside the holes are also typable. Furthermore, we can establish that the types of the subexpressions in the holes are complementary in the sense of the relations given in the Figures 4.5 and 4.6. If the two subexpressions in the holes are replaced with expressions of "shorter", in the sense of the relation given in the Figure 4.11, but still complementary types and with lower joint effects, the typability of the entire expression will be preserved, with decrement of its effects.

Lemma 4.5.6 (Replacement $\mathbb{D}_r$) If $\Gamma \vdash \mathbb{D}_r[P, Q] : U : :^i[T]_\kappa$, then there exist some $U_1, T_1, \kappa_1$ and $U_2, T_2, \kappa_2$ such that $\Gamma \vdash P : U_1 : :^1[T]_\kappa$, and $\Gamma \vdash Q : U_2 : :^2[T]_\kappa$ and, depending on the polarity, either $T_1 \leq T_2$ or $T_2 \leq T_1$ and if there exist $P', Q'$ such that $\Gamma \vdash P' : U_3 : :^3[T]_\kappa$, and $\Gamma \vdash Q' : U_4 : :^4[T]_\kappa$ and $U_3 \leq U_1$, $T_3 \leq T_1$, $U_4 \leq U_2$, $T_4 \leq T_2$, $\kappa_3 \cup \kappa_4 \subseteq \kappa_1 \cup \kappa_2$ and $T_3 \leq T_2$ (if $T_1 \leq T_2$) or $T_4 \leq T_3$ (if $T_2 \leq T_1$), then $\Gamma \vdash \mathbb{D}_r[P', Q'] : U' : :^i[T]_\kappa'$, for some $i, \kappa'$ such that $U' \leq U$, $T' \leq T$, $i' \subseteq i$, $\kappa' \subseteq \kappa$.

Proof: By induction on the structure of $\mathbb{D}_r$ and by compositionality of the type system.

Theorem 4.5.7 (Subject Reduction) If $\Gamma \vdash_S P : U : :^i[T]_\kappa$ and $P \rightarrow P'$, then there exist $U', i', T', \kappa'$ such that $U' \leq U$, $T' \leq T$, $i' \subseteq i$, $\kappa' \subseteq \kappa$ and $\Gamma \vdash_S P' : U' : :^i'[T']_\kappa'$.

Proof: By case analysis of the reduction rules given in Figure 4.2. Thanks to Lemmas 4.5.4, 4.5.5 and 4.5.6 it is enough to show that the processes in the contexts before and after the reduction are typeable as prescribed by these lemmas.

Case (END).

Firstly, we consider the case in which the l.h.s. has been typed using rule (LevSign):

$$
\Gamma \vdash 0 : \text{end} : :^i[\text{end}]_{\perp} \quad \text{(LevSign)}
$$

for the typing of the r.h.s. we use the rule (Nil):

$$
\Gamma \vdash_\emptyset 0 : \text{end} : :^i[\text{end}]_{\ell} \quad \text{(Nil)}
$$
Similarly, if the l.h.s. has been typed using rule \((\text{LevSignRT})\), we have:

\[
\Gamma \vdash 0 : \text{end} :: \perp [\text{end}]_{\ell}
\]

\[
\Gamma \vdash \ell \triangleleft 0 : \text{end} :: \perp [\text{end}]_{\ell}
\]

(\text{LevSignRT})

and the typing of the r.h.s. is:

\[
\Gamma \vdash 0 : \text{end} :: \perp [\text{end}]_{\ell}
\]

(\text{Nil})

\textbf{Case (Inv).} Firstly, we consider the case in which the service invocation has been typed using rule \((\text{LevSign})\). In this case the processes building the redex in the l.h.s. are typed as follows:

\[
\Gamma \vdash P \in \text{end} :: \perp [T]_{\kappa}\quad \Gamma \vdash s : [T']_{j}, T' \leq T
\]

\[
\Gamma \vdash \sigma.P : \text{end} :: \perp [\text{end}]_{\kappa \cup j} \quad \text{(Inv)}
\]

\[
\kappa \cup j \subseteq \ell
\]

\[
\Gamma \vdash \theta \ell \triangleleft \sigma.P : \text{end} :: \perp [\text{end}]_{\ell}
\]

\[
\Gamma \vdash Q : \text{end} :: \perp [T']_{k_1} \quad \Gamma \vdash s : [T']_{j}, k_1 \subseteq j
\]

\[
\Gamma \vdash (s).s.Q : \text{end} :: \perp [\text{end}]_{k_1} \quad \text{(ServDef)}
\]

Then, we can type the changed processes in the r.h.s. as follows:

\[
\Gamma \vdash P : \text{end} :: \perp [T]_{\kappa}\quad \Gamma \vdash \ell \triangleleft P : \text{end} :: \perp [T]_{\ell}\quad T' \leq T
\]

\[
\Gamma \vdash \theta \ell \triangleleft P : \text{end} :: \perp [\text{end}]_{\ell}
\]

(\text{LevSign})

\[
\Gamma, r : [T']_{\ell} \vdash r^{-} \triangleright \ell \triangleleft P : \text{end} :: \perp [\text{end}]_{\ell}
\]

\[
\Gamma, r : [T']_{\ell} \vdash \theta r^{-} \triangleright \ell \triangleleft P : \text{end} :: \perp [\text{end}]_{\ell}
\]

(\text{ToTop})

\[
\Gamma \vdash Q : \text{end} :: \perp [T']_{k_1}\quad k_1 \subseteq \ell
\]

\[
\Gamma \vdash \ell \triangleleft Q : \text{end} :: \perp [T']_{\ell}
\]

(\text{LevSignRT})

\[
\Gamma, r : [T']_{\ell} \vdash r^{+} \triangleright \ell \triangleleft Q : \text{end} :: \perp [\text{end}]_{\ell}
\]

\[
\Gamma, r : [T']_{\ell} \vdash r^{+} \triangleright \ell \triangleleft Q : \text{end} :: \perp [\text{end}]_{\ell}
\]

(\text{Sess})

\[
\Gamma, r : [T']_{\ell} \vdash \theta r^{+} \triangleright \ell \triangleleft Q : \text{end} :: \perp [\text{end}]_{\ell}
\]

(\text{ToTop})

\[
\Gamma, r : [T']_{\ell} \vdash r^{+} \triangleright \ell \triangleleft Q : \text{end} :: \perp [\text{end}]_{\ell}
\]

\[
\Gamma, r : [T']_{\ell} \vdash \theta r^{+} \triangleright \ell \triangleleft Q : \text{end} :: \perp [\text{end}]_{\ell}
\]

(\text{TopPar})

so:

\[
\Gamma, r : [T']_{\ell} \vdash r^{+} \triangleright \ell \triangleleft Q : \text{end} :: \perp [\text{end}]_{\ell}\quad \Gamma, r : [T']_{\ell} \vdash (s).s.Q : \text{end} :: \perp [\text{end}]_{k_1}
\]

\[
\Gamma, r : [T']_{\ell} \vdash \theta r^{+} \triangleright \ell \triangleleft Q : \text{end} :: \perp [\text{end}]_{\ell}
\]

(\text{TopPar})

Notice that the typing of the restriction for \(r\) will discharge the premise \(r : [T']_{\ell}\).

We will now consider the case in which the service invocation has been typed using rule \((\text{LevSignRT})\).

\[
\Gamma \vdash P : U :: [T]_{\kappa}\quad \Gamma \vdash s : [T']_{j}, T' \leq T
\]

\[
\Gamma \vdash \sigma.P : \text{end} :: \perp [U]_{\kappa \cup j \cup \ell}
\]

\[
\Gamma \vdash \ell \triangleleft \sigma.P : \text{end} :: \perp [U]_{\ell}
\]

(\text{LevSignRT})
The typing of service definition is clearly as it was in the previous case, so we can derive \( \Gamma, r : [T]_r \vdash_{\{s\}} r^+ \triangleright \ell \ | \ s.Q : \text{end} : \vdash^{+}[\text{end}]_e \). Moreover we can derive:

\[
\begin{align*}
\Gamma & \vdash P : U ::[T]_\kappa & \kappa \sqcup i \subseteq \ell \quad (\text{LevSignRT}) \\
\Gamma & \vdash \ell \triangleleft P : U ::[T]_\ell & T' \leq T \\
\Gamma, r : [T]_r \vdash r^+ \triangleright \ell \triangleright \triangleright P : \text{end} ::[U]_e \\
\end{align*}
\]

\( \triangleleft \Gamma \)

**Case (COM).** By Lemma 4.5.6 the processes which build the redex in the l.h.s. must be typed as follows:

\[
\begin{align*}
\Gamma, x : S_\ell & \vdash P : U ::[T]_\kappa & \Gamma \vdash v : S_\ell \\
(\text{Out}) \\
\end{align*}
\]

\( \triangleleft \Gamma \)

\( \vdash (x).P : U ::[?(S_\ell).T]_{\kappa \ell \ell} \)

\( \Gamma, x : S_\ell \vdash Q : U' ::[T']_{\kappa'} \\
\text{pipe}(U ::[?(S_\ell).T]_{\kappa \ell \ell}, U' ::[T']_{\kappa'} S_\ell) = U'' ::[T'']_{\kappa''} \\
\)

\( \vdash (v).P > x > Q : U'' ::[T'']_{\kappa''} \\
\) (Pipe)

We distinguish two cases:

1. \( T = \text{end} \). In this case the resulting type of \( (v).P \) is \( U \circ U' ::[!\text{end}]_{\kappa \ell \ell \ell} \), and the type of \( P > x > Q \) is \( U ::[\text{end}]_\kappa \) (the final case of the function pipe). Then by Lemma 4.5.3 we can type the r.h.s. as follows:

\[
\begin{align*}
\Gamma \vdash P > x > Q : U ::[\text{end}]_\kappa & \quad \Gamma \vdash Q[\{x\}] : U' ::[T']_{\kappa'} \\
\Gamma \vdash P > x > Q \mid Q[\{x\}] : U \circ U' ::[!\text{end}]_{\kappa \ell \ell \ell} & \quad (\text{Par}) \\
\end{align*}
\]

2. \( T = !\text{end}h \), \( h \geq 1 \). In this case \( !\text{end}h \), the resulting type of \( (v).P \) is \( U \circ U' ::[\text{end}]_{\kappa \ell \ell \ell} \), and the type of \( P > x > Q \) is \( U \circ U' ::[\text{end}]_{\kappa \ell \ell \ell} \) (the second case of the function pipe). Then by Lemma 4.5.3 we can type the r.h.s. as follows:

\[
\begin{align*}
\Gamma \vdash P > x > Q : U \circ U' : \vdash_{\kappa \ell \ell \ell} [\text{end}]_{\kappa \ell \ell \ell} & \quad \Gamma \vdash Q[\{x\}] : U' ::[T']_{\kappa'} \\
\Gamma \vdash P > x > Q \mid Q[\{x\}] : (U \circ U') \circ U' ::[\text{end}]_{\kappa \ell \ell \ell} & \quad (\text{Par}) \\
\end{align*}
\]

We conclude observing that \( (U \circ U') \circ U' = U \circ U'^{h+1} \).

**Case (LEVSEL).** We only consider the case in which the l.h.s. has been typed using rule (LevSignRT), the case in which rule (LevSign) has been used being similar and simpler. From the typing of the l.h.s.:

\[
\begin{align*}
\Gamma & \vdash P_i : U ::[T]_{\ell_i} & \kappa_i \Subset \ell_i \quad (1 \leq i \leq n) \\
\exists j (1 \leq j \leq n). \ell_1 \sqcap \cdots \sqcap \ell_n = \ell_j \\
\end{align*}
\]

\( \triangleleft \Gamma \)

\[
\begin{align*}
\Gamma & \vdash \bigcup_{i=1}^{n} \ell_i \times P_i : U ::[w]_{\ell_1, \ldots, \ell_n} \times T[\ell_j] \\
\end{align*}
\]

\( \triangleleft \Gamma \)

\[
\begin{align*}
\Gamma & \vdash \ell \triangleleft P_i \times P : U ::[w]_{\ell_1, \ldots, \ell_n} \times T[\ell] \\
\end{align*}
\]

\( \triangleleft \Gamma \)

\( \vdash (w).P : U ::[w]_{\ell_1, \ldots, \ell_n} \times T[\ell] \)
and the condition \( \iota \cup \ell_i \subseteq \ell \) we get \( \iota \cup \kappa_i \subseteq \ell \) and then we can type the r.h.s. as follows:

\[
\frac{\Gamma \vdash P_i : U \lll T_{\kappa_i} \quad \iota \cup \kappa_i \subseteq \ell}{\Gamma \vdash \ell \triangle P_i : U \lll T_{\ell}} \quad (\text{LevSignRT})
\]

### 4.5.2 Security Properties

From an access control point of view, clients are regarded as “subjects”, and services and basic values as “objects”. Therefore, a client is required to possess an appropriate clearance to activate a service or to communicate a value. On the other hand, services are regarded as trusted entities, and they are allowed to act on behalf of the client who invoked them (this is reflected in the operational semantics, where the body process of the service definition “inherits” the clearance from the client after the activation).

A process can be active (or running) only in the scope of a security framing. In fact, the reduction rules from Figure 4.2 can be redefined with explicit notion of reduction under the security frame, and decorating

**Figure 4.12: Labeled Operational Semantic Rules**

**Figure 4.13: Security Errors**
the values with their security levels. This reduction relation is given in Figure 4.12.

This allows a straightforward definition of \textit{security error}, given in Figure 4.13. A process runs in a security error in the following cases:

1. when it tries to activate a service with a level not below its clearance;
2. when it tries to communicate a value with a level not below its clearance in the current session;
3. when it tries to return a value with a level not below its clearance or the parent clearance toward the parent session;
4. when it tries to communicate a value with a level not below its clearance to a pipe or not below the parent clearance toward a pipe in the parent session;
5. when it uses values with a level not below its clearance in the control expression of conditional branching;
6. when it enters into the level branching where all the levels are not below its clearance.

This allows us to express the main result of this chapter as a type safety property in the following theorem.

\textbf{Theorem 4.5.8} Typable processes do not run into security errors.

\textbf{Proof:} by contradiction and case analysis of the reduction rules given in Figure 4.13. In each case we show the subdeductions which assure the required order between security levels. We only consider processes typed using rule \textbf{(LevSignRT)}, the case in which rule \textbf{(LevSign)} has been used being similar and simpler.

\textbf{Case (InvErr):}

\begin{align*}
\Gamma \vdash P : U \vdash [T]_{\kappa} \quad \Gamma \vdash \nu_{\ell'} : [\overline{T}]_{\ell'} \quad T' \leq T \\
\Gamma \vdash \nu.P : \text{end} \vdash [\overline{T}]_{\kappa \cup \ell \cup \nu \subseteq \ell} \quad \text{(Inv)} \quad \kappa \cup \ell' \cup \nu \subseteq \ell \\
\Gamma \vdash \ell' \triangleleft \nu.P : \text{end} \vdash [\overline{T}]_{\kappa}
\end{align*}

\textbf{Case (ComErr):}

\begin{align*}
\Gamma, x_i : S_i \vdash P : U \vdash [T]_{\kappa} \quad (\text{Inp}) \quad \kappa \cup \ell \cup \nu \subseteq \ell \\
\Gamma \vdash (x_i).P : U \vdash \nu([S_i].T)_{\kappa \cup \nu} \quad (\text{LevSignRT}) \\
\Gamma \vdash \ell' \triangleleft (x_i).P : U \vdash \nu([S_i].T)_{\kappa \cup \nu} \quad (\text{Sess})
\end{align*}

where $T' = \begin{cases} T, & \text{if } p = + \\ T', & \text{if } p = - \end{cases}$. We get $\nu \subseteq \kappa \cup \ell \cup \nu \subseteq \ell$.

\textbf{Case (RetErr):} As in previous case the typing of $\nu.P : \ell' \triangleleft (x_i).P$ assures $\nu \subseteq \ell'$. Moreover we have:

\begin{align*}
\Gamma \vdash Q : U \vdash [T]_{\kappa} \quad \Gamma \vdash v_i : S_i \quad (\text{Ret}) \quad \nu \cup \kappa \subseteq \ell \\
\Gamma \vdash \text{return } v_i.Q : (S_i).U \vdash [T]_{\kappa} \quad (\text{LevSignRT}) \\
\Gamma \vdash \ell' \triangleleft \text{return } v_i.Q : (S_i).U \vdash [T]_{\kappa} \quad (\text{Sess}) \\
\Gamma, r : [T']_{\ell} \vdash r.P : \ell' \triangleleft \text{return } v_i.Q : \text{end} \vdash [T]_{\kappa}
\end{align*}
where $T' = \begin{cases} T, & \text{if } p = + \\ T', & \text{if } p = - \end{cases}$, which implies $\iota \subseteq \iota \cup \kappa \subseteq \ell$.

**Case (PipeErr):**

\[
\begin{align*}
\Gamma \vdash P : U :: \lambda |(S_i)^\eta_{\kappa,\ell} & \quad \Gamma \vdash \nu : S_i \\
\vdash (\text{Out}) & \quad \Gamma, \nu_1 : S_i \vdash Q : U' :: \lambda |T'|_{\kappa,\ell} \\
\Gamma \vdash (\nu_1) P > x > Q & \quad U' :: \lambda |T'|_{\kappa,\ell} \\
\Gamma \vdash (\nu_1) P > x > Q & \quad U'' :: \lambda |T''|_{\kappa,\kappa',\ell} \\
\end{align*}
\]

where $\text{pipe}(U : \lambda |(S_i)^{n-1}{\text{end}}_{\kappa,\ell}, U' : \lambda |T'|_{\kappa,\ell}) = U'' :: \lambda |T''|_{\kappa,\kappa',\ell}$, which implies $\iota \subseteq j \cup j' \cup \kappa \cup i \cup \kappa' \subseteq \ell$.

**Case (PipeRetErr):** The reasoning is similar to that of cases (RetErr) and (PipeErr).

**Case (IfErr):**

\[
\begin{align*}
\Gamma \vdash \nu_1 : S_i \quad \Gamma \vdash P : U :: \lambda |T|_{\kappa,\ell} & \quad \Gamma \vdash Q : U :: \lambda |T|_{\kappa,\ell} \\
\vdash (\text{if}) & \quad \iota \subseteq \kappa \cap \kappa' \subseteq \ell \\
\Gamma \vdash \nu_1 = \nu_2 & \quad \text{then } P \text{ else } Q : U :: \lambda |T|_{\kappa,\ell} \\
\end{align*}
\]

Note that $\iota \subseteq \kappa \cap \kappa' \subseteq \iota \cup \kappa \cup \kappa' \subseteq \ell$.

**Case (LevSelErr):**

\[
\begin{align*}
\Gamma \vdash P_1 : U :: \lambda |T|_{\kappa,\ell} & \quad \kappa_1 \subseteq \iota_i \quad (1 \leq i \leq n) \\
\exists j \quad (1 \leq j \leq n) & \quad \ell_j \cap \cdots \cap \ell_n = \ell_j \\
\Gamma \vdash \prod_{i=1}^n \ell_i \times P_1 : U :: |T|_{\kappa,\ell} & \quad \text{(LevSel)} \\
\Gamma \vdash \ell \times \prod_{i=1}^n \ell_i \times P_1 : U :: |T|_{\kappa,\ell} & \quad \text{(LevSignRT)} \\
\end{align*}
\]

Note that $\ell_i \subseteq \ell_j$.

4.6 Progress

This section discusses the progress property. The proof of this property is similar to the one given in [12], taking into account that the security constructs and annotations do not lead to stuck states by Theorem 4.5.8. We start with some definitions.

We first define the initial and final processes (or normal forms) of the reductions we will consider. We start reductions from top level closed processes which do not offer communications and do not have public services.

**Definition 4.6.1 (Initial Processes)** A process $P$ is initial if $\emptyset \vdash \emptyset : \text{end} :: \lambda |\text{end}|_{\kappa}$ without using the runtime typing rules.

Our final configurations are just parallels of service definitions.

**Definition 4.6.2 (Normal Forms)** A process $P$ is in normal form if $P \equiv (\nu s_1) \ldots (\nu s_n) \prod_{i=1}^n (s_i, Q_i)$.

It is also useful to name processes where session calls are not ready to reduce.

**Definition 4.6.3 (Call-free Processes)** A process $P$ is call-free if $P$ does not contain redexes which are session calls.
We classify session names according to the processes they enclose and their relative positions. A meaningful session name enclose processes which can be reduced.

**Definition 4.6.4 (Meaningful Session Names)** A session name $r$ is meaningful in a process $P$ if $P$ contains $r^P \triangleright Q$ with $Q \neq 0$.

An innermost session name is a meaningful session name which does not contain meaningful session names.

**Definition 4.6.5 (Innermost Session Names)** A meaningful session name $r$ is innermost in a process $P$ if $P \equiv D_r[Q,R]$ and $Q,R$ do not contain meaningful session names.

Our definition of progress property is standard.

**Definition 4.6.6 (Progress Property)** A process $P$ has the progress property if $P \rightarrow^* Q$ implies that either $Q$ is reducible or $Q$ is in normal form.

The following lemma says that session calls can always be satisfied in a well-typed process.

**Lemma 4.6.7** If $\emptyset \vdash_0 (\nu s) C[\ell \triangleright s.P] : \text{end}$, then $C[\ell \triangleright s.P] \equiv (\nu m) s.Q | R$ for some $m, Q, R$ with $s \notin m$.

**Proof:** Rule (Inv) is necessary to type $s.P$ and it requires an assumption for the service name $s$, which can only be discharged by rule (ServRestr). In turn rule (ServRestr) implies that the service name $s$ decorates the turn-style, and so rule (ServDef) for the service name $s$ needs to have been applied.

It is crucial to observe that starting from an initial process we always get processes where all session names occur exactly twice with opposite polarities and at least one session name, if any, is innermost.

**Lemma 4.6.8** If $P$ is an initial process and $P \rightarrow^* (\nu m) Q$, where $Q$ does not contain restrictions, then

1. if $r^P$ occurs in $Q$, then it occurs once and also $r^\neg P$ occurs once in $Q$, and
2. either no session name occurs in $Q$ or there is at least one session name which is innermost in $Q$.

**Proof:** The proof is by cases on the reduction rules applied.

Rule (INV) is the only reduction rule which creates a meaningful session name, and it creates two occurrences of an innermost session name, with opposite polarities, since the processes in the scope of the new name cannot contain session names.

The last but one structural rule cancels both polarities of a session name, and so rule (STR) preserves the property that each session name occurs twice with both polarities. The rules for structural equivalence only deal with non meaningful session names, so rule (STR) does not modify the innermost session names in a process.

The last lemma says that under suitable conditions a process offering a communication inside the innermost session has always a partner accepting the communication.

**Lemma 4.6.9** Let $P$ be an initial process and $P \rightarrow^* (\nu m) D_r[Q,R]$, where $r$ is innermost in $D_r[Q,R]$ and $D_r[Q,R]$ does not contain restrictions, cannot be reduced by rules (Pipe), (PipeRet), (IfT), (IFF), (LevSel) and it is call-free. Then
1. $Q \equiv (x).Q'$ implies $R \equiv \langle v \rangle.R'$;
2. $Q \equiv \langle v \rangle.Q'$ implies $R \equiv (x).R'$;
3. $Q \equiv \text{return } v.Q'$ implies $R \equiv \langle v \rangle.R'$;
4. $Q \equiv \sum_{i=1}^{n}(l_i).Q_i$ implies $R \equiv \langle l_k \rangle.R'$;
5. $Q \equiv \langle l_k \rangle.Q'$ implies $R \equiv \sum_{i=1}^{n}(l_i).R_i$.

**Proof:** The typability of $\mathbb{D}_r[[Q, R]]$ implies by Lemma 4.5.6 that $Q, R$ are typed with session types such that one is a subtype of the dual of the other, and since they are irreducible by hypothesis $R$ must have the shapes shown in cases 1, 2, 4 and 5. In case 3 since $\mathbb{D}_r[[Q, R]]$ is typed by $\text{end } \downarrow \text{end}$, the process $Q$ must be enclosed in the body of another session, that we can call $r_1$, getting $\mathbb{D}_r[[Q, R]] \equiv \mathbb{D}_r[[P_1, C_r P[Q]]]$. By Lemma 4.5.6 the processes $P_1, C_r P[Q]$ are typed with session types such that one is a subtype of the dual of the other, and this gives $P_1 \equiv (x).P'$.

We are now able to prove the progress theorem.

**Theorem 4.6.10 (Progress)** If $P$ is initial, then $P$ has the progress property.

**Proof:** If $P$ is not in normal form, then $P$ must contain some process different from service definition. If $P \equiv C[\ell \leftarrow Q]$, then $P$ is reducible by Lemma 4.6.7. If $P \equiv C[R > x > Q]$, then either $R$ can be reduced or typability implies $R \equiv \langle v \rangle.R'$ or $R \equiv C_r P[\text{return } v.R']$, so rule (PIPE) or (PIPERET) is applicable. If $P \equiv C[\text{if } u = v \text{ then } R \text{ else } Q]$, then either rule (IFT) or rule (IFF) is applicable. If $P \equiv C[\ell \leftarrow \bigcup_{i=1}^{n} \ell_i \times P_i]$, then rule (LEVSEL) is applicable by Theorem 4.5.8.

Otherwise $P \equiv C[Q]$ where $Q$ is an input, output, value return, label choice or label-guarded sum. Since $P$ is typed by $\text{end } \downarrow \text{end}$, and $P$ is call-free, then the process $Q$ must be enclosed in the body of a session. In this case we can choose an innermost session and apply Lemma 4.6.9 to conclude that $P$ is reducible.
Chapter 5

Conclusion

In the first part of this thesis, we have shown a way of integrating access control and information flow control in the setting of a high-level programming language, involving a declassification construct. Our “state-oriented” approach, that we share with [5], differs from the “value-oriented” approach (that one has to adopt when dealing with purely functional languages) that is followed in [19, 33, 38] to deal with stack inspection, and [32, 37] as regards information flow control (see also [34] for further references).

We think that assuming that confidentiality levels are assigned to “information containers” is more in line with the usual way of dealing with confidentiality than assigning security levels to values, like boolean $tt$ and $ff$, integers or functions. In this way, the safety property guaranteed by access control is quite simple and natural, and this also provides a natural restriction on the use of declassification, and, more generally, information flow.

The second part of this thesis addressed security issues in a calculus for services and pipelines (CaSPiS) by employing static analysis access control methodology, and by offering increased modularity of the services with options of different behaviours of a service depending on the privileges of the client who activated it. The type system ensures at compile time that a service can be activated only by clients with appropriate clearance, while preserving the safety and progress properties of previous versions of CaSPiS.

The approach in this work was to consider mandatory access control, with no mechanisms of access right elevation or demotion. A next step might be to consider discretionary access control with a possibility of permission passing among subjects.
Bibliography


