Typing Secure Information Flow: Declassification and Mobility

Ana Almeida Matos

Typing Secure Information Flow: Declassification and Mobility

Ana Almeida Matos

	Ph.D. Degree
Confered by	École Nationale Supérieure des Mines de Paris
Type	Doctorat Européen
Subject	Informatique temps réel, robotique et automatique
Promotors	Gérard Boudol, INRIA Sophia Antipolis
	Ilaria Castellani, INRIA Sophia Antipolis
	Defense
Date	January 31st, 2006
Place	Sophia Antipolis, France
Jury	Prof. David Sands, Chalmers University of Technology and University of Göteborg
	Prof. Michele Bugliesi, Università Ca' Foscari
	Prof. Roberto Amadio, Université Paris 7
	Prof. Vasco Vasconcelos, Universidade de Lisboa

Titre
Typage du flux d'information sûr : déclassification et mobilité
Mots clés
sécurité, flux d'information, systèmes de types,
déclassification, mobilité
Resumé
voir page xix
Synthèse
voir page xxi

Funded by
Portuguese Ministry of Science and Technology
Fundação para a Ciência e Tecnologia (FCT)
POSI/SFRH/BD/7100/2001

Hosts
INRIA Sophia Antipolis, France (main host)
Chalmers University of Technology, Sweden
University of Brighton, England
University of Twente, Netherlands

To my beloved ones.

caminante, no hay camino,
se hace camino al andar.

Antonio Machado

Acknowledgments

I am truly grateful to my supervisors Gérard Boudol and Ilaria Castellani for the availability, guidance and support that they offered me during the last four years. I hope to have learned from their scientific insight and rigor, as well as from their many other qualities, which have become a personal reference to me.

I am also thankful to the Mimosa group and INRIA Sophia Antipolis for providing me excellent working conditions. Sincere recognition goes to the Portuguese Foundation for Science and Technology (FCT), which generously supported my research and studies for four years, in this inspiring place, between the Alps and the Mediterranean.

Part of my doctoral studies took place in three one-month research visits to three major European Universities. I would like to thank Andrei Sabelfeld, and the ProSec group, for having kindly welcomed me during an exciting research experience at Chalmers Institute of Technology. Special thanks go also to Matthew Hennessy, and the Foundations of Computation group of the University of Brighton, for orientation and stimulation on my first steps doing independent research. Thanks are also due to Mariëlle Stoelinga, for having offered me ideal conditions, at the University of Twente, for a good start in the quest of writing the present thesis.

Let me now mention Jan Cederquist, Tamara Rezk, Sylvain Schmitz and other unnamed researchers around the world, who decisively contributed to enhance my research and career with fruitful discussions, reviews or advice.

Finally, I would like to express my warm gratitude to all those, family and friends (and pets!), who have nourished my mind and heart during the past times with plenty of encouragement and love. I have always felt their presence, in happy and harder times, right here in France, but also all the way from Portugal, or even from as far away as Laos!

Obrigada!

Contents

List of Figures

Abstract

We address the issue of confidentiality and declassification in a language-based security approach. We study, in particular, the use of refined type and effect systems for statically enforcing flexible information flow policies over imperative higher-order languages with concurrency. A general methodology for defining and proving the soundness of the type and effect system with respect to such properties is presented. We consider two main topics:

- The long-standing issue of finding a flexible information control mechanism that enables declassification. Our declassification mechanism takes the form of a local flow policy declaration that implements a local information flow policy.
- The largely unexplored topic of controlling information flow in a global computing setting. Our network model, which naturally generalizes the local setting, includes a notion of domain, and a standard migration primitive for code and resources. New forms of security leaks that are introduced by code mobility are revealed.

In both the above settings, to take into account dynamic flow policies we introduce generalizations of non-interference, respectively named the *non-disclosure* and the *non-disclosure for networks* policies. Their implementation is supported by a concrete presentation of the security lattice, where confidentiality levels are sets of principals, similar to access control lists.

Résumé (en français)

Nous nous intéressons au sujet de la confidentialité et de la déclassification. Nous étudions en particulier l'usage d'un système de types et d'effets pour assurer de manière statique des politiques de sécurité flexibles pour un langage d'ordre superieur impératif avec concurrence. Une méthodologie générale pour définir et prouver la correction du système de types et d'effets pour de telles proprietés est présentée. Nous considérons deux points principaux :

- La question de trouver un mécanisme flexible de contrôle d'information qui permet la déclassification. Notre mécanisme de déclassification prend la forme d'une déclaration de politique locale de flux qui implémente une politique locale de flux d'information.
- La question jusqu'ici inexplorée de contrôler les flux d'information dans un environnement global. Notre modèle de réseau, qui généralise l'environnement global, inclut une notion de domaine et une primitive de migration standard pour le code et les ressources. De nouvelles formes de perte d'information, introduites par la mobilité du code, sont révélées.

Dans les deux cas mentionnés ci-dessus, pour prendre en compte les politiques de flux globales nous introduisons des généralisations de la non-interférence, qui sont nommées *non-divulgation* et *non-divulgation pour les réseaux*. Ces généralisations sont obtenues à l'aide d'une représentation concrète des treillis de sécurité, où les nivaux de confidentialité sont des ensembles de principaux, semblables à des listes de contrôle d'accès.

Synthèse (en français)

Motivation

Depuis l'invention des ordinateurs, le problème de la sécurité informatique a pris une importance croissante. Un de ses principaux objectifs est la *confidentialité*, c'est-à-dire l'assurance que seules les entités autorisées puissent accéder à l'information.

Les premiers efforts datent des balbutiements de l'informatique, aboutissant à partitionner la mémoire et à assurer que les programmes en cours d'exécution ne puissent accéder aux partitions des autres programmes. Cela constitue un des premiers exemples de *contrôle d'accès*, une forme de protection de la confidentialité qui comprend un mécanisme permettant au système d'autoriser ou d'interdire l'accès à certaines données et l'exécution de certaines actions. Cependant, une fois une autorisation délivrée, le contrôle d'accès n'est pas en mesure de réguler la propagation de l'information dévoilée pendant l'exécution d'un programme [Denning, 1976 ; Lampson, 1973 ; Myers et Liskov, 2000 ; Sabelfeld et Myers, 2003]. Cette observation a suscité une attention croissante pour le *contrôle du flux d'information*. Son but est précisément de savoir et de maîtriser comment l'information circule dans un système informatique, afin d'en interdire l'accès aux entités non autorisées.

En quelques décades seulement, les systèmes d'information ont évolué de machines travaillant en temps partagé vers des réseaux mondiaux d'ordinateurs, où programmes et données transitent de manière décentralisée. Dans cet environnement d'*informatique globale*, les problèmes de sécurité sont devenus cruciaux. En effet, les nouvelles possibilités offertes par la globalisation ont souvent été exploitées à des fins douteuses (virus, vers, refus de service *etc.*). Étonnamment, très peu d'études ont été réalisées sur le contrôle du flux d'information dans les réseaux. C'est le sujet de cette thèse.

Typage du flux d'information sûr

Nous adoptons une *approche langage* (voir [Sabelfeld et Myers, 2003] pour un état de l'art), ce qui signifie que nous restreignons notre attention aux flux d'information qui ont lieu lors de l'exécution de programmes. De ce fait, les fuites d'information ne peuvent se produire que lors d'un transfert entre les objets d'un langage donné.

Afin de spécifier quels échanges d'information sont acceptables, il est naturel d'attribuer des *niveaux de sécurité* aux objets (données et canaux), qui ne pourront alors être lus que par des sujets dotés d'accréditations correspondantes. Une

relation d'ordre est donnée pour ces niveaux d'accréditation [Denning, 1976], ce qui signifie que, au cours de l'exécution, l'information est autorisée à transiter d'un objet à l'autre si l'objet source a un niveau de sécurité plus faible que l'objet cible. La relation d'ordre sur les niveaux de sécurité détermine les flux légaux. Ceci a été défini formellement en premier lieu par la notion de *dépendance forte* par [Cohen, 1977], et est aussi connu sous le nom de *non-interférence* dans la terminologie utilisée par [Goguen et Meseguer, 1982].

Un travail considérable a été consacré à la conception de méthodes permettant d'analyser les flux d'information dans un programme (voir par exemple [Andrews et Reitman, 1980] pour les premières références). L'analyse peut être effectuée dynamiquement, en utilisant des vérifications à l'exécution. Ces méthodes sont critiquables de par leurs coûts importants en calcul et en mémoire, et aussi parce qu'elles peuvent dévoiler de l'information du fait même de l'échec d'une vérification à l'exécution [Denning, 1976 ; Myers et Liskov, 1997]. D'un autre côté, des méthodes d'analyse statique des flux d'information ont été développées, permettant de rejeter les programmes non sûrs avant leur exécution. On retiendra en particulier l'emploi de systèmes de types, qui a débuté avec le travail de Volpano, Smith et Irvine [Volpano *et al.*, 1996]. Bien qu'ils n'offrent qu'une analyse approximative, les systèmes de types décidables ont des avantages reconnus, tels que la prévention d'erreurs de programmation. Des systèmes de types qui assurent des flux d'information sécurisés ont été mis au point pour de nombreux langages (par exemple [Boudol et Castellani, 2002 ; Crary *et al.*, 2005 ; Heintze et Riecke, 1998 ; Pottier et Simonet, 2003 ; Smith, 2001 ; Smith et Volpano, 1998 ; Volpano et Smith, 1997 ; Volpano *et al.*, 1996 ; Zdancewic et Myers, 2002], et d'autres références dans [Sabelfeld et Myers, 2003]), incluant des langages réalistes tels que Jif (ou JFlow, voir [Myers, 1999]) et Flow Caml [Simonet, 2003].

Problèmes abordés

> « En dépit de leur longue histoire et de leurs qualités, les mécanismes de contrôle des flux d'information n'ont pas encore été utilisés avec succès en pratique. »
>
> [Zdancewic, 2004]

Précision

Une grande part des efforts consacrés au contrôle des flux d'information consiste à déterminer quels sont les échanges d'information non désirés. Même en choisissant l'objectif le plus strict, et en s'efforçant de rejeter toutes les fuites d'information sécurisée, il reste encore un travail de longue haleine avant de pouvoir décider quelles sortes d'échanges d'information pourraient être exploitées dangereusement. Ce problème est fortement corrélé au degré d'expressivité du contexte dans lequel le programme est exécuté. Typiquement, lorsque l'on introduit de nouvelles fonctionnalités dans un langage de programmation, de nouvelles formes de fuites d'information apparaissent. Il y donc un besoin réel pour des analyses de sécurité sur des langages qui sont au moins aussi expressifs que ceux utilisés en pratique. Dans cette thèse nous baserons notre étude sur Core ML [Milner *et al.*, 1997 ; Wright et Felleisen, 1994], un λ-calcul en appel

par valeur et des constructions impératives que nous enrichissons encore avec des processus légers concurrents.

Il est aisé de trouver des mécanismes pour sélectionner uniquement les programmes sûrs : un exemple extrême serait de n'en sélectionner aucun. Valider autant de programmes sûrs que possible est un tout autre problème. En fait, déterminer si un programme est sûr est souvent indécidable, ce qui rend les procédures de rejet des programmes non sûrs forcément excessives. Lors du développement de systèmes de types pour les flux d'information, la clef semble être d'identifier les effets des programmes et les niveaux de sécurité de l'information dont ces effets dépendent. En formalisant la notion d'*effet* de manière de plus en plus détaillée, on devient à même d'exprimer des conditions de plus en plus précises pour accepter les programmes. Cela est montré dans cette thèse en considérant un système de types et effets [Lucassen et Gifford, 1988] qui comprend des effets de lecture, écriture et de terminaison de programmes.

Flexibilité

Il est intéressant de noter que, même dans des systèmes où la sécurité a une importance cruciale, le plus souvent la non-interférence n'est pas la politique choisie. En effet, le rejet aveugle de toute possibilité de fuite d'information empêcherait le fonctionnement de programmes qui sont très courants et très utiles. Des programmes de vérification de mots de passe ou de chiffrage sont des exemples typiques, pour lesquels le principe même implique la déclassification (même très partielle) d'une information secrète pour des observateurs publics. La non-interférence est donc prohibitivement restrictive pour une utilisation pratique. Cet état de fait a récemment motivé la recherche de propriétés de sécurité alternatives, plus flexibles que la non-interférence et permettant une forme de déclassification ou une autre (voir [Volpano, 2000 ; Volpano et Smith, 2000 ; Myers *et al.*, 2004 ; Sabelfeld et Myers, 2004 ; Chong et Myers, 2004 ; Mantel et Sands, 2004 ; Li et Zdancewic, 2005] et une synthèse dans [Sabelfeld et Sands, 2005]). Cependant, la plupart de ces approches sont influencées par la crainte de voir la déclassification, une fois permise, être utilisée pour délivrer plus d'information que ce qui est considéré comme sûr. Il en résulte que les propriétés de sécurité existantes dans la littérature incluent souvent des restrictions qui diminuent leur aptitude à remplacer la non-interférence.

Dans cette thèse, nous soutenons que, avant de prévoir des restrictions sur les utilisations de la déclassification, l'on devrait fournir des moyens flexibles et simples d'exprimer cette dernière, et nous proposons la *non-divulgation* comme une généralisation naturelle de la non-interférence. En particulier, il serait bon de pouvoir exprimer délibérément des opérations qui impliquent des flux d'information rejetés par la relation d'ordre sur les niveaux de sécurité. À cette fin, nous proposons un mécanisme pour étendre localement la relation d'ordre qui régule les flux permis grâce à une *déclaration de flux*. Cela permet au programmeur de configurer la politique de sécurité pour chaque situation particulière à l'aide de simples conditions de flux sur les niveaux de sécurité.

Intégration

Afin de construire des applications réelles pour la sécurité des flux d'information, il est nécessaire de l'intégrer avec les mécanismes de sécurité existants.

Comme mentionné plus tôt, l'articulation entre contrôle d'accès et contrôle de flux d'information est particulièrement importante. Le contrôle d'accès est typiquement assuré par les systèmes d'exploitation grâce à des *listes de contrôle d'accès* (listes d'entités autorisées). Les systèmes de contrôle de flux peuvent être spécifiés en termes concrets de labels de sécurité [Myers et Liskov, 1997 ; Banerjee et Naumann, 2005]. Dans cette thèse, nous franchissons une étape supplémentaire, et nous spécifions nos politiques de sécurité en termes de ces entités. Plus précisément, nos déclarations de flux concernent directement les relations de flux entre entités, à partir desquelles les relations d'ordre sur les niveaux de sécurité peuvent être dérivées. De cette manière nous suggérons que contrôle d'accès et contrôle flexible de flux d'information peuvent être simplement combinés.

À un niveau supérieur, protéger la confidentialité des données est un problème particulièrement sensible dans un contexte d'informatique globale. Quand information et programmes transitent au travers de réseaux, ils sont exposés à des utilisateurs avec différents intérêts, buts et responsabilités. Ceci motive la recherche de mécanismes pratiques qui assurent le respect de la confidentialité des informations, tout en minimisant le besoin de se contenter d'une confiance mutuelle. Dans cette thèse nous présentons une première étude sur les flux d'information non sûrs introduits par la mobilité dans le contexte d'un langage distribué avec des états. La pertinence de l'idée selon laquelle l'informatique globale amène de nouvelles difficultés dans le domaine de l'analyse de flux est confirmée, puisque nous avons identifié une nouvelle forme de faille de sécurité, la *faille de migration*, qui peut apparaître dans des environnements distribués permettant la mobilité.

Contributions

Les contributions au cœur de cette thèse sont :

- L'étude et le développement de systèmes de types et d'effets implémentant la sécurité des flux d'information pour des langages basés sur un lambda-calcul impératif d'ordre supérieur avec processus légers et création de références.
- L'introduction d'une déclaration de flux permettant la déclassification. La présentation d'une politique de sécurité qui est une généralisation directe de la non-interférence : la politique de non-divulgation. Un nouveau système de types et d'effets correct intégrant cette propriété. Cette contribution est basée sur les travaux publiés dans [Almeida Matos et Boudol, 2005] et (en ce qui concerne le système de types) dans [Almeida Matos, 2005].
- L'identification d'une nouvelle faille de sécurité qui apparaît dans un environnement impératif quand la mobilité des ressources joue un rôle explicite. La formulation et la formalisation d'une propriété de sécurité, non-divulgation dans un réseau, qui permet la déclassification dans un environnement distribué permettant de la mobilité. Un système de types et d'effets correct permettant d'assurer cette propriété. Cette contribution est basée sur [Almeida Matos, 2005].

Contenu de la thèse

Dans les trois chapitres de cette thèse, nous considérons un environnement de plus en plus complexe. Le chapitre 2 est consacré à l'étude de la *non-interférence* dans un langage concurrent d'ordre supérieur, le chapitre 3 à la possibilité d'une *déclassification* dans le langage, et le chapitre 4 à un environnement permettant la mobilité et la déclassification. Dans chacun de ces chapitres, les propriétés de sécurité des flux d'information sont étudiées, formalisées et assurées par le biais de systèmes de types.

Non-interférence en environnement concurrent

Le chapitre 2 aborde le problème de la confidentialité dans un environnement concurrent simple. Son objectif est d'introduire la *non-interférence* comme une propriété qui détermine l'absence d'échanges non sûrs d'information, et d'illustrer l'utilisation d'un système de types et d'effets pour assurer cette propriété. Pour cela, nous considérons un langage concurrent expressif : un lambda-calcul impératif d'ordre supérieur avec processus légers et références. Nous utilisons une forme particulière de treillis de sécurité, basés sur l'idée d'*entités*, qui peuvent être paramétrées par une *politique globale des flux*. Nous étudions comment contrôler les failles de sécurité qui peuvent apparaître dans ce langage, en particulier les *failles de terminaison* et les *failles d'ordre supérieur*. Le contexte considéré ici servira de base aux développements présentés tout au long de cette thèse.

Ce chapitre est organisé comme suit. Dans la première section, nous introduisons quelques idées de base sur l'utilisation de systèmes de types pour rejeter des flux d'information considérés non sûrs sur le plan de la non-interférence. Dans la section 2.2, nous définissons un lambda-calcul d'ordre supérieur qui présente des problèmes de confidentialité dus à l'environnement concurrent simple. Dans la section 2.3, nous définissons un pré-treillis de sécurité basé sur les entités, et nous donnons une définition de la non-interférence adaptée à l'environnement concurrent à partir d'une bisimulation. Dans la section 2.4, nous développons un système de types qui accepte seulement des programmes avec cette propriété. Les propriétés de base du langage et du système de types, dont la correction, sont données dans cette section. Enfin, nous analysons les travaux liés.

Non-divulgation et déclassification

Dans le chapitre 3 nous abordons le problème de la déclassification. L'intention est d'introduire une déclaration de flux qui étend localement la politique globale de flux dans un λ-calcul d'ordre supérieur impératif avec processus légers et références, fournissant ainsi un mécanisme permettant d'exprimer la déclassification dans la portée de la déclaration. Pour cela, nous proposons la *non-divulgation* comme une propriété qui généralise la non-interférence et qui permet la déclassification, et nous démontrons comment employer un système de types et d'effets pour garantir cette politique. Cette vue dynamique des politiques de flux d'information est mise en place grâce au pré-treillis de sécurité basé sur les entités présenté au début de la section 2.3. Nous étudions quelles formes de flux d'information sont permises par la politique de non-divulgation mais interdites par une politique de non-interférence.

Ce chapitre est organisé comme suit. Dans la première section nous justifions le besoin d'exprimer la déclassification, et introduisons la déclaration de flux comme un outil qui peut être employé sous le contrôle d'une politique de non-divulgation. Dans la section 3.2, nous présentons une extension au langage d'ordre supérieur impératif de la section 2.2, auquel nous ajoutons une instruction de déclaration de flux. Puis, dans la section 3.3, nous introduisons notre généralisation de la non-interférence, c'est-à-dire la politique de non-divulgation, qui prend en compte des politiques de contrôle de flux dynamiques. Un système de types et d'effets est donné pour le langage dans la section 3.4, ainsi que certaines propriétés de base de ce système, dont la correction. Nous discutons ensuite de travaux similaires.

Non-divulgation pour du code mobile

Le chapitre 4 aborde les problèmes de confidentialité et de déclassification en informatique globale d'un point de vue orienté langage. L'objectif est de gérer les nouvelles formes de failles de sécurité introduites par la mobilité du code, que nous appelons *failles de migration*. Nous présentons une généralisation de la politique de non-divulgation de [Almeida Matos et Boudol, 2005] pour les réseaux, et un système de types et d'effets pour l'assurer. Nous considérons le même langage que dans le chapitre précédent, enrichi par une notion de domaine d'exécution et d'une primitive de migration standard.

Ce chapitre est organisé comme suit. Dans la première section, nous définissons un calcul qui permet d'exprimer les problèmes qui apparaissent avec les communications en réseau. Dans la section 4.3, nous étudions l'introduction de politiques multiples pour les flux, et nous formulons une propriété de non-divulgation utilisable dans un environnement décentralisé. Dans la section 4.4, nous développons un système de types qui n'accepte que les programmes vérifiant cette propriété. Enfin, nous étudions des travaux similaires.

Les explications techniques de chacun de ces chapitres sont clairement délimitées, et peuvent être lues indépendamment. Cependant, les langages, les propriétés de sécurité, et les systèmes de types qui sont présentés dans les derniers chapitres sont construits sur ceux des chapitres précédents. Par la suite, des remarques sur ce processus incrémental sont faites. Naturellement, les explications dans les derniers chapitres sont plus avancées, se concentrant sur les nouvelles fonctionnalités introduites. Les preuves des chapitres 2 et 3 peuvent être aisément reconstruites à partir de celles du chapitre 4, qui peut être vu comme une généralisation des deux autres.

Conclusion

Contributions principales et travaux futurs

Nous résumons à présent les principales contributions techniques de cette thèse, et nous donnons quelques perspectives sur les suites possibles à lui donner. Nous finissons par quelques remarques sur les principaux points introduits dans cette thèse.

Politiques de sécurité Nous avons abordé le problème de savoir quels programmes sont sûrs sur le plan de la confidentialité dans les flux d'information. Nous avons étudié deux politiques de sécurité majeures : la propriété de non-interférence classique, qui détermine l'absence de transferts d'information non sûrs grâce à un ordre statique et global sur les niveaux de sécurité, et une propriété nouvelle dite de non-divulgation, qui détermine l'absence d'échanges d'information non sûrs grâce à un ordre dynamique de niveaux de sécurité valide à ce point.

Les politiques de non-interférence et de non-divulgation ont été définies en termes de bisimulations, basées naturellement sur des changements d'état atomiques. Ceci offre la précision nécessaire pour pouvoir d'une part analyser les changements en mémoire qui se produisent à chaque étape de l'exécution (analyse nécessaire dans un environnement concurrent), et d'autre part pour indiquer la politique de flux valide (nécessaire pour restreindre la portée des déclarations de flux) en étiquetant la sémantique des changements atomiques. Nous montrons que les relations d'ordre sur les niveaux de sécurité peuvent être exprimées comme de simples politiques de flux, comme les relations entre entités.

Nous pensons que la non-divulgation est une généralisation naturelle de la non-interférence classique, et que l'idée d'utiliser une bisimulation sur des sémantiques de changements atomiques étiquetées pour définir une politique de sécurité reflétant la nature *locale* de la déclassification pourrait peut-être être utilisée dans d'autres cadres. Par exemple, en nous inspirant de la remarque de Biba selon laquelle l'intégrité est en quelque sorte le dual de la confidentialité (voir [Li *et al.*, 2003 ; Myers et Liskov, 1997]), nous pourrions mettre au point un cadre théorique similaire pour les aspects d'intégrité des données de la sécurité, en incluant potentiellement des fonctionnalités de dégradation comme l'« *endorse* »de [Li *et al.*, 2003 ; Myers *et al.*, 2004].

Paradigmes Nous avons orienté nos recherches en direction de deux paradigmes concurrents. L'un est dans un environnement local, où les processus légers peuvent être créés dynamiquement et exécutés en parallèle sur le même support de calcul. L'autre est dans environnement distribué avec migration des processus, où les processus légers sont exécutés dans différents domaines, et où les localisations relatives des processus et des ressources déterminent les circonstances dans lesquelles ils peuvent être exécutés. Dans ce dernier paradigme, nous avons découvert que de nouvelles formes de failles de sécurité, les failles de migration, peuvent être encodées. Nous avons trouvé certaines ressemblances avec les fuites d'informations causées par la localisation des processus dans un réseau de type "Ambient" [Crafa *et al.*]. En effet, dans cet article, la visibilité des processus légers dans un réseau est aussi considérée comme sujette aux contraintes de confidentialité. Cela indiquerait que nos résultats ne sont pas confinés à notre modèle de réseau particulier.

Nous avons volontairement choisi un modèle simple de la mobilité, cependant suffisant pour présenter les principes sous-jacents des failles de migration[1].

[1]Le choix de lier statiquement les références aux processus légers, plutôt qu'aux domaines [Ravara *et al.*] est justifié par le fait que, dans ce dernier cas, le problème de la confidentialité des données migrées en même temps que les processus ne se pose pas. De plus, ce cas peut être imité dans notre modèle en attribuant chacune des références d'un domaine à un processus léger fixé dans ce domaine, et en ne permettant qu'aux processus légers sans références d'être déplacés d'un domaine à l'autre.

Néanmoins, on peut s'attendre à voir des modèles plus complexes de l'informatique globale avoir des effets intéressants sur l'étude des contrôles de flux d'information. Par exemple, la possibilité d'avoir une forme plus générale de migration qui peut être provoquée par un processus léger sur un autre (migration objective) va probablement amener de nouveaux moyens d'exprimer des failles de migration. D'un autre côté, introduire un calcul de conformité [Boudol, 2005a] comme prérequis pour laisser entrer un processus dans un domaine pourrait fournir de nouveaux moyens de prévenir les fuites d'information.

Un axe de recherche prééminent pour les modèles d'informatique globale est consacré à la nature peu fiable des réseaux. Comme indiqué dans [Boudol, 2004], les principes des systèmes réactifs semblent particulièrement adaptés pour fournir des réactions aux erreurs. Le langage ULM qui y est présenté montre comment cela peut être réalisé. Il pourrait être intéressant de voir quel impact l'intégration de ces principes réactifs pourrait avoir dans le modèle utilisé pour cette thèse.

Fonctionnalités des langages Les langages sur lesquels nous avons basé notre étude sont simples mais expressifs. Notre point de départ était un lambda-calcul d'ordre supérieur impératif avec création de processus légers et de références. Ce noyau a ensuite été enrichi avec une déclaration de flux qui permet de paramétrer dynamiquement la relation d'ordre sur les niveaux de sécurité. Une version du langage utilisant la notion de domaine d'exécution a enfin été considérée, avec une instruction de migration d'un processus léger et de ses références.

Notre mécanisme de déclassification, la déclaration de flux, peut être rendue encore plus expressif. Il serait particulièrement intéressant d'étendre le langage de manière à utiliser des niveaux de sécurité de première classe [Tse et Zdancewic, 2004 ; Zheng et Myers, 2004]. De plus, l'idée de permettre d'introduire des politiques de flux dynamiquement peut certainement être appliquée à d'autres paradigmes de programmation. Réciproquement, on pourrait imaginer d'autres moyens de restreindre l'usage de déclarations de flux, et adapter alors la politique de non-divulgation en conséquence.

Mécanismes de certification Pour appliquer les politiques de sécurité sur les programmes de nos langages, nous avons présenté de nouveaux systèmes de types et d'effets qui sont motivés par des principes assez similaires. En particulier, celui du chapitre 3 propose une variante de [Almeida Matos et Boudol, 2005] qui restreint la déclassification à n'être possible que par le biais d'opérations de déclassification contenues dans une déclaration de flux. Nous avons donc mis en évidence la distinction entre notre nouveau paradigme de déclassification et la déclassification plus classique par dégradation de valeur.

Les preuves de correction des systèmes de types, qui sont expliquées en détail pour les langages distribués et pour la politique de non-divulgation, sont elles aussi très semblables. Nous avons de bonnes raisons de penser que le mécanisme de preuve de correction de type, qui est basé sur celui de [Almeida Matos et Boudol, 2005], (qui à son tour est basé sur celui de [Boudol et Castellani, 2002]), peut aussi être appliqué à d'autres cas.

Des généralisations immédiates pour le système de types et d'effets pourraient être l'introduction de polymorphisme et d'inférence de type [Myers,

1999 ; Pottier et Simonet, 2003]. Des raffinements supplémentaires pourraient être potentiellement obtenus en considérant un ensemble d'effets plus riche, comme par exemple la création et la destruction de références, la création de processus légers, et plus généralement toute action modifiant le contexte d'une expression dans la machine (abstraite) qui l'évalue.

Remarques finales

La déclaration de flux : encore un mécanisme de déclassification ?

Comme vu dans les discussions du chapitre 3 liées aux travaux similaires (voir section 3.5), les propositions pour des mécanismes de déclassification fleurissent dans la littérature. Plutôt que de seulement proposer encore un mécanisme de déclassification, nous avons suggére un moyen d'affronter « la difficulté [de] déterminer quelle nature un mécanisme de déclassification devrait avoir et quel genre de garanties de sécurité il permettrait »[Zdancewic, 2004]. L'idée clef est que, avant de réfléchir à comment contrôler l'usage de la déclassification, il serait bon de disposer d'un cadre théorique pour l'exprimer. Nous pensons que le cadre théorique pour la déclassification que nous avons présenté dans cette thèse est séduisant pour les raisons suivantes :

- Il fournit un mécanisme simple, mais flexible et puissant pour la déclassification. En particulier, il ne comporte pas de restrictions allant au-delà de la simple déclassification.
- Il inclut une politique de sécurité, la non-divulgation, avec des propriétés sémantiques satisfaisantes. Nous remarquons que deux de ces propriétés font partie de celles suggérées par le « bon sens »pour les politiques de sécurité dans [Sabelfeld et Sands, 2005] : la *cohérence sémantique*, qui voudrait que les programmes qui sont sémantiquement équivalents soient uniformément classés comme sûrs ou non sûrs, et la *monotonicité de la sécurité*, qui voudrait que d'une part la non-divulgation soit équivalente à la non-interférence pour les programmes sans déclassification, et que d'autre part les programmes ne deviennent pas non sûrs par le simple ajout de déclarations de flux.
- Il est aisément extensible à d'autres langages et environnements. En particulier, notre politique de non-divulgation est *extensionnelle*, c'est-à-dire définie en termes de sémantique de programme, indépendamment des particularités du langage.
- Il fournit une technique fiable pour rejeter avec une précision raisonnable tous les programmes qui ne respectent pas la politique de sécurité.

La première des qualités ci-dessus est peut-être la plus importante de notre cadre théorique pour la déclassification. En effet, nos déclarations de flux peuvent exprimer la déclassification avec n'importe quel degré de précision, depuis des opérations spécifiques jusqu'à des portions entières d'un programme, et ce entre n'importe quels niveaux de sécurité. Ceci est rendu possible par la manipulation directe des politiques de contrôle des flux, qui sont de simples relations binaires entre les entités du système.

Nous notons finalement que, en incorporant notre mécanisme de déclassification dans notre étude des contrôles de flux d'information pour réseaux, nous avons montré sa fiabilité dans un nouvel environnement de calcul.

Sur la combinaison de la déclassification et de la mobilité

Les thèmes de la déclassification et de la mobilité dans les flux d'information sont des problèmes assez indépendants. Il n'est peut-être pas si surprenant qu'ils puissent être combinés aussi aisément. Cependant, nous devons préciser que cette facilité est due à la nature hautement décentralisée des déclarations de flux. Aucun accord global n'est présupposé sur les politiques de déclassification (comme c'est le cas dans [Mantel et Sands, 2004]). De plus, les changements de la politique de gestion des flux qui sont opérés dynamiquement par les programmes ont une portée lexicale, et n'affectent pas le système dans son ensemble.

Les dangers potentiels nés de la déclassification dans un environnement permettant la mobilité du code paraissent peut-être plus frappants que ses avantages. On peut imaginer l'exemple d'un processus léger en cours de migration contrôlé par une politique de flux très permissive : quand il arrive dans un domaine où un autre processus comportant des références secrètes est en cours d'exécution, il pourrait déclassifier cette information, indépendamment de la politique de flux de leur propriétaire. Ceci pourrait être exprimé dans notre langage comme :

$$d_1[(\text{goto } d_2); (\text{flow } H \prec L \text{ in } (m.b_L :=^? (?\ n.a_H)))^m] \ \parallel \ d_2[N^n] \qquad (1)$$

Mesuré à l'aune de la non-divulgation pour les réseaux, ce programme est sûr ; en fait, le processus léger m respecte la politique de flux déclarée quand il copie la valeur de la référence $n.a_H$ dans $m.b_L$. Cependant, on peut voir le potentiel de formuler d'autres politiques de sécurité qui tiennent compte de la propriété des informations, ou le potentiel de définir des constructions du langage qui conditionnent l'exécution de sous programmes à des politiques de flux plus strictes, ou même le potentiel de mettre au point des conditions de type pare-feux qui contrôlent l'entrée de processus légers mobiles.

Réciproquement, nous nous tournons vers l'exemple d'un processus léger qui amène ses propres données dans un site où il devra effectuer des calculs en privé. Alors, la possibilité de déclarer ses propres politiques de sécurité se transforme en un avantage. Par exemple, nous pourrions écrire le programme

$$d_1[(\text{goto } d_2); (\text{flow } H \prec L \text{ in } (n.b_L :=^? (?\ m.a_H)))^m] \ \parallel \ d_2[N^n]. \qquad (2)$$

Nous n'avons encore que peu d'expérience pratique dans l'utilisation de systèmes informatiques mobiles, ce qui rend délicat d'évaluer l'importance de permettre la déclassification dans un environnement mobile. Néanmoins, la déclassification semble être une fonctionnalité cruciale pour tout langage sujet à un contrôle de flux d'information, ce qui justifie *a fortiori* son inclusion dans le langage mobile du chapitre 4. De plus, afin d'évaluer les problèmes et avantages apportés, il est préférable d'étudier la déclassification sur un langage simple mais expressif, qui peut ensuite être utilisé comme fondation sur laquelle construire des cadres théoriques plus complexes. Nous pensons que le langage mobile présenté dans cette thèse est un point de départ fertile pour l'étude des flux d'information sécurisés dans les réseaux.

Chapter 1

Introduction

1.1 Motivation

Computer security has been an increasingly important matter since computers have existed. One of its main concerns is *confidentiality*, the assurance that information is accessible only by those who are authorized.

First efforts date back to the early times of computing, leading to the definition of memory partitions and ensuring that running programs do not access partitions of other programs. This is an early example of *access control*, an aspect of confidentiality enforcement that refers to any mechanism by which a system grants or revokes the right to access some data, or perform some action. However, once a reading clearance has been given, it is beyond the reach of access control to regulate the propagation of the released information as it is being processed by a program [Denning, 1976; Lampson, 1973; Myers & Liskov, 2000; Sabelfeld & Myers, 2003]. This observation has led to increased attention on the control of information flow. The aim of *information flow control* is precisely to track and regulate how information flows in a computing system, to prevent it from falling into the hands of unauthorized parties.

In just a few decades, computational systems have evolved from local time-sharing machines to complex world wide webs of computing devices, where programs and data roam in a decentralized fashion. In such a *global computing* environment, security issues become particularly crucial. Indeed, the new possibilities offered by global computing have been often exploited by parties with hazardous intentions (think of viruses, worms, denial of service attacks etc.). Surprisingly, very little work has been done on the control of information flow in networks. This is the general topic of the present thesis.

1.1.1 Typing Secure Information Flow

We take a *language based approach* (see [Sabelfeld & Myers, 2003] for a review), which means that we restrict our attention to information flows that take place within computations of programs. Hence, information leaks can only occur as information is transferred between the computational objects of a given language.

In order to specify which are the allowed information flows, it is natural to attribute security levels to objects (information containers or channels), to

be read only by subjects with the corresponding security clearance. Then, an ordering relation is given for these security levels [Denning, 1976], meaning that, during computation, information is allowed to flow from one object to another only if the source object has a lower security level than the target one. That is, the ordering relation on security levels determines the legal flows, and a program is secure if, when executing, it never performs illegal flows. This was first formally stated via a notion of *strong dependency* by [Cohen, 1977], and was later captured by the notion of *non-interference* [Goguen & Meseguer, 1982].

A considerable amount of work has been devoted to the design of methods for analyzing information flow in programs (see for instance [Andrews & Reitman, 1980] for early references). The analysis can be done dynamically, using run-time checks. These methods can be criticized for involving an important computational and storage overhead, or for leaking information by the mere failure of a run-time check [Denning, 1976; Myers & Liskov, 1997]. As an alternative, static analysis methods have been developed for information flow, allowing rejection of insecure programs before execution. One can highlight the use of type systems, which started with the work of Volpano, Smith and Irvine [Volpano *et al.*, 1996]. Although they offer only an approximate analysis, (decidable) type systems have well-known advantages, like preventing some programming errors at an early stage. Type systems that enforce secure information flow have been designed for various languages (e.g. [Boudol & Castellani, 2002; Crary *et al.*, 2005; Heintze & Riecke, 1998; Pottier & Simonet, 2003; Smith, 2001; Smith & Volpano, 1998; Volpano & Smith, 1997; Volpano *et al.*, 1996; Zdancewic & Myers, 2002], and further references in [Sabelfeld & Myers, 2003]), including for full-fledged languages like Jif (or JFlow, see [Myers, 1999]) and Flow Caml [Simonet, 2003].

1.1.2 Addressed Challenges

> "Despite their long history and appealing strengths, information-flow mechanisms have not yet been successfully applied in practice."
>
> [Zdancewic, 2004]

Refinement

Much of the effort in controlling information flow goes into pinning down which are the unwanted information flows. Even when the most conservative goal is chosen, and one strives for strictly rejecting every security leak, there is still a long way to go in understanding which forms of information flow could potentially be exploited harmfully. This issue is strongly related to the expressivity of the computational context in which programs run. Typically, when introducing new features in a programming language, new forms of security leaks appear. There is thus a genuine requirement of performing security analysis for languages that are at least as expressive as the ones that are used in practice. In this thesis we shall base our study on Core ML [Milner *et al.*, 1997; Wright & Felleisen, 1994], a call-by-value λ-calculus extended with imperative constructs that we further enrich with concurrent threads.

It is easy to find mechanisms for selecting only secure programs – an extreme example could be the selection of no program at all. A different matter is to have

them validate as many secure programs as possible. In fact, security of programs is often undecidable, which makes procedures for rejecting insecure programs necessarily excessive. In the design of type systems for information flow, the key seems to lie in identifying the effects of the programs and the security levels of the information that those effects depend on. By formalizing the notion of *effect* in increasingly detailed manners, one can express increasingly refined conditions for accepting programs. This is exemplified in this thesis, by considering a type and effect system [Lucassen & Gifford, 1988] that deals with reading, writing and termination effects of programs.

Flexibility

It is interesting that, even in systems where security is of crucial importance, most often non-interference is not the desired policy. Indeed, the blind rejection of any information leak would disallow programs that are quite common and very useful. Typical examples are password checkers and encryption programs, whose whole purpose implies the *declassification* of secret information (even if only by a bit) to public observers. Non-interference is thus prohibitively restrictive to be used in practice. This has recently motivated a lot of research in alternative security properties that are more flexible than non-interference and allow some kind of declassification (see [Volpano, 2000; Volpano & Smith, 2000; Myers *et al.*, 2004; Sabelfeld & Myers, 2004; Chong & Myers, 2004; Mantel & Sands, 2004; Li & Zdancewic, 2005] and an overview in [Sabelfeld & Sands, 2005]). However, most of the approaches are influenced by the concern that, once allowed, declassification could be misused into leaking more information than what should be considered "safe". As a result, security properties that exist in the literature often include built-in restrictions which impair their suitability for replacing non-interference.

In this thesis we argue that, prior to envisioning restrictions on the usages of declassification, one should provide flexible and simple means to express it, and we propose *non-disclosure* as a natural generalization of non-interference. In particular, one would like to have the possibility of expressing operations that deliberately involve flows of information that are rejected by the underlying security ordering. To this end, we provide a mechanism for locally extending the security ordering that regulates the allowed flows by means of a *flow declaration* construct. This allows the programmer to customize the security policy to the particular application that she/he has in mind, by means of simple flow conditions on the security levels.

Integration

In order to build real applications for information flow security, one must show how to integrate it with existing security mechanisms. As we have mentioned in the beginning, the interface between access control and information flow control is intrinsically relevant. Access control is typically enforced by operating systems by means of *access control lists* (lists of authorized principals). Information-flow systems can be specified in terms of such concrete forms of security labels [Myers & Liskov, 1997; Banerjee & Naumann, 2005]. In this thesis, we go a step further, and we specify our security policies also in terms of principals. In particular, our flow declarations deal directly with flow relations between principals, from

which ordering relations on the security levels can be derived. In this way we suggest that access control and flexible information flow control can be combined in a simple manner.

At a higher level, protecting confidentiality of data is a concern of particular relevance in a global computing context. When information and programs move throughout networks, they are exposed to users with different interests, goals and responsibilities. This motivates the search for practical mechanisms that enforce the respect for confidentiality of information, while minimizing the need to rely on mutual trust. In this thesis we present a first study on insecure information flows that are introduced by *mobility* in the context of a distributed language with states. Substantiating the pertinence that the global computing setting brings new issues into information flow analysis, we have identified a new form of security leaks, the *migration leaks*, which can appear in distributed settings with mobility.

1.2 Overview

1.2.1 Structure of the Thesis

In the following three chapters, we consider increasingly complex computational settings. Chapter 2 focuses on the study of *non-interference* in a *higher-order concurrent* language, Chapter 3 on the admittance of *declassification* in the language, and Chapter 4 on a setting with mobility and declassification. In each of these chapters, information flow security properties are studied, formalized, and enforced by means of type systems.

Chapter 2 introduces the fundamental notion of Non-interference as a property that states the inability of secure programs to allow information leaks to occur during their computations. We consider the particular context of an imperative higher-order λ-calculus with thread creation. Given the concurrent setting in which computations take place, Non-interference is formalized in terms of a bisimulation that uses a small-step semantics for the language. A type and effect system for accepting secure programs is given, and its soundness proof is outlined.

Chapter 3 motivates the need for more flexibility in the rejection of programs where information leaks occur. By introducing a way of changing what is considered to be an illegal information flow (leak) it is possible to give the programmer means to declassify information. For this purpose a flow declaration construct is added to the language considered in Chapter 2. A new security policy, called Non-disclosure, which generalizes Non-interference, is formulated. A type and effect system that generalizes the one in Chapter 2 is presented, and its soundness proof is outlined.

Chapter 4 turns to the topic of global computing and the information flow security problems that are specific to this context. The language in Chapter 3 is enlarged so as to reflect the distributed notion of computations, as well as to allow for the mobility of code and resources. New security leaks that arise are identified in a generalization of Non-disclosure that we call Decentralized Non-disclosure. A more complex type and effect system,

which generalizes the one in Chapter 3, is presented as well. A soundness proof is given in detail.

The technical expositions of each of the above chapters are self-contained, and can be read independently to a large extent. However, the languages, security properties and type systems that are presented in the latter chapters build upon the preceding ones. Therefore, remarks underlining this constructive process will be made. Naturally, explanations in latter chapters will be more advanced, focusing on the new features that are introduced. The proofs of Chapters 2 and 3 can be easily reconstructed from the one in Chapter 4, which can be seen as a generalization of the two.

1.2.2 Contributions

The central contributions of this thesis are:

- The study of the design of type and effect systems implementing information flow security, for languages based on an imperative higher-order λ-calculus with thread and reference creation.
- The introduction of a flow declaration construct for the purpose of declassification. The presentation of a security policy that is a direct generalization of Non-interference – the Non-disclosure policy. A new sound type and effect system for enforcing such a property. This contribution is based on the published work [Almeida Matos & Boudol, 2005] and (as regards the type system) [Almeida Matos, 2005].
- The identification of new security leaks that arise in an imperative setting where mobility of resources plays an explicit role. The formulation and formalization of a security property – Non-disclosure for Networks – that allows for declassification in a distributed setting with mobility. A sound type and effect system for enforcing such a property. This contribution is based on the published work [Almeida Matos, 2005].

Chapter 2

Non-interference in Concurrency

This chapter addresses the issue of confidentiality for a simple concurrent setting. The purpose is to introduce *non-interference* as a property that determines the absence of insecure information flows, and to illustrate the use of a type and effect system for enforcing that property. To this end, we consider an expressive concurrent language: an imperative (higher-order) lambda-calculus with thread and reference creation. We use a specific kind of security lattice, based on the idea of *principals*, that can be parameterized by a *global flow policy*. We study how to control security leaks that arise in programs of this language, including *termination leaks* and *higher-order leaks*. The context that we consider here will serve as basis for the developments presented in this thesis.

The chapter is organized as follows. In the next section we introduce some basic ideas behind using type systems to reject information flows that are considered insecure from the point of view of non-interference. In Section 2.2 we define a higher-order calculus that exhibits confidentiality problems that arise in simple concurrent settings. In Section 2.3 we define a principal-based security pre-lattice, and give a bisimulation-based definition of non-interference that is suitable for a concurrent setting. In Section 2.4 we develop a type system that only accepts programs satisfying such a property. Basic properties of the language and of the type system, including soundness, are given in this section. Finally, we discuss related work.

2.1 Introduction

We informally explain the non-interference property, and give examples of different forms of security leaks, culminating in the *higher-order leaks*. We provide intuitions on how to use type systems and security lattices to rule out insecure information flows. We introduce the concepts of principal and flow policy, and show how they can be used to give a concrete interpretation of security levels.

2.1.1 Basics of Non-interference

Let us briefly recall the intuition about non-interference in a system with only two security levels, *low* (public, L) and *high* (secret, H). These security levels are attributed to objects, meaning that the information contained in them can only be read by subjects with the corresponding security clearance. Informally, the non-interference property states that information should only be allowed to flow from lower to higher (more secure) levels. To start with, let us consider a sequential imperative language, such as the imperative language of Volpano, Smith and Irvine [Volpano *et al.*, 1996]. The objects of the system are variables, whose security levels are specified using subscripts (for example a_H is a variable of high security level).

Direct Leaks and Control Leaks

An *insecure flow* of information, or *interference*, can be said to occur when the initial values of high variables influence the final value of low variables. The simplest case of insecure flow is that of an assignment of the value of a high variable to a low variable, as in:

$$b_L := a_H \tag{2.1}$$

It is called an explicit insecure flow, and consists of a *direct leak* of information. More subtle kinds of flow, called implicit flows, may be induced by the flow of control (*control leaks*), as in the program

$$\text{if } a_H = tt \text{ then } b_L := tt \text{ else } b_L := ff \tag{2.2}$$

where at the end of execution the value of b_L may give information about a_H. A similar program can be written using a loop:

$$b_L := ff; (\text{while } a_H = tt \text{ do } (b_L := tt; a_H := ff)) \tag{2.3}$$

Other programs may be considered as secure or not depending on the context in which they might appear. For instance, the program

$$(\text{while } a_H = tt \text{ do nil}); b_L := ff \tag{2.4}$$

may be considered safe in our sequential setting (since whenever it terminates it produces the same value ff for b_L), whereas it may become critical in the presence of parallelism, as we shall see next.

Termination Leaks

In a sequential setting it makes sense to look only at the output values that a program may give, thus ignoring all its non-terminating computations (we refer to this form of non-interference as "basic"). In fact, it is only the output values of terminating computations that can be used by other programs that are sequentially composed with that program. Furthermore, if a computation enters a non-terminating loop, it is not possible for other programs to interrupt the loop since they will never get their turn to execute. This is no longer true in a concurrent setting. In fact, Example 2.4 can be used in a program that

always terminates and is insecure in the above sense. In the following program, suppose that variables c_H and c'_H are initially assigned the value ff and let $||$ be the parallel operator:

$$\begin{array}{c} \text{if } a_H \text{ then } c_H := tt \text{ else } c'_H := tt \;||\\ (\text{while } \neg c_H \text{ do nil}); b_L := ff; c'_H := tt \;||\\ (\text{while } \neg c'_H \text{ do nil}); b_L := tt; c_H := tt \end{array} \tag{2.5}$$

This program always terminates, and the final value of b_L reflects the initial value of the high variable a_H. The insecure flow that occurs here is usually called a *termination leak*, since it results from the "termination behavior" of a portion of the program.

Higher-Order Leaks

The language that we will consider in this chapter is *higher-order*, which intuitively means that programs can be stored in the memory of another program and executed from it. This enables the emergence of another sort of security leaks, hereby called *higher-order leaks*. To illustrate them let us consider a very rudimentary form of higher-order expressiveness: we assume that variables that contain programs and are written after a '$*$' are executable expressions that simply return their contents. Then, if the program M is placed in the variable c, the program $* c$ behaves as the program M. This allows us to write the following program

$$(\text{if } a_H \text{ then } c_H := (b_L := tt) \text{ else } c_H := (b_L := ff)); * c_H \tag{2.6}$$

which is insecure in that it has the same effect as the program of Example 2.2.

We will come back to the treatment of higher-order security leaks in Section 2.4.

2.1.2 Typing Away Security Leaks

In order to reject programs that can exhibit dangerous flows of information, rules of thumb are conceived to simplify the problem of detecting where they might occur. Typically they state that programs cannot contain:

- Assignments of high information to low variables (Example 2.1).
- Low assignments in the body of conditionals or loops with high tests (Examples 2.2 and 2.3).

In a concurrent setting, one additionally wants to reject programs containing:

- Low assignments following conditionals or loops with high tests (Example 2.4).

Rules like the above can be used to build type systems for rejecting insecure programs before their execution. But it is important to understand that these syntactic rules can only approximate the semantic notion of secure program. When they are used to reject all insecure programs, some secure ones are necessarily left out too.

Sequential Setting

Typing rules for sequential programs might look like this:

$$\frac{\vdash e : \delta \quad \delta \leq \theta}{\vdash (a_\theta := e) : \theta}$$
$$\frac{\vdash e : \delta \quad \vdash P : \theta_1 \quad \vdash Q : \theta_2 \quad \delta \leq \theta_1, \theta_2}{\vdash (\text{if } e \text{ then } P \text{ else } Q) : \theta_1 \wedge \theta_2} \tag{2.7}$$
$$\frac{\vdash e : \delta \quad \vdash P : \theta \quad \delta \leq \theta}{\vdash (\text{while } e \text{ do } P) : \theta}$$

In the above rules, similar to those in [Volpano *et al.*, 1996], the greek letters δ and θ represent security levels. In the simple setting we have considered so far, they can be either H or L, where the relation $L \leq H$ means that L is less secret or equal to H. The relation $\leq$ is called a *flow relation*, since it establishes the direction in which information may flow. The operator $\wedge$ gives the smallest of two security levels, that is $L \wedge L = L$, $L \wedge H = L$ and $H \wedge H = H$. Judgments $\vdash e : \delta$ mean that expression e has reading effect δ – it represents an upper-bound to the security levels of the variables that are needed to calculate e. Judgments $\vdash P : \theta$ mean that the program P has the *writing effect* θ – it represents a lower-bound to the security levels of the variables that are written in P. In this way, the condition $\delta \leq \theta$ in the first rule implies that a high expression cannot be assigned to a low variable. On the other hand, the same conditions in the second and third rules imply that there cannot be a low write in P if e is a high test.

Concurrent Setting

In a concurrent setting, low assignments following conditionals or loops with high tests can be rejected using rules like the following:

$$\frac{\vdash e : \delta \quad \delta \leq \theta}{\vdash (a_\theta := e) : (\theta, \sigma)}$$
$$\frac{\vdash e : \delta \quad \vdash P : (\theta_1, \sigma) \quad \vdash Q : (\theta_2, \sigma) \quad \delta \leq \theta_1, \theta_2}{\vdash (\text{if } e \text{ then } P \text{ else } Q) : (\theta_1 \wedge \theta_2, \delta \vee \sigma)} \tag{2.8}$$
$$\frac{\vdash e : \delta \quad \vdash P : (\theta, \sigma) \quad \delta \vee \sigma \leq \theta}{\vdash (\text{while } e \text{ do } P) : (\theta, \delta \vee \sigma)}$$
$$\frac{\vdash Q_1 : (\theta_1, \sigma_1) \quad \vdash Q_2 : (\theta_2, \sigma_2) \quad \sigma_1 \leq \theta_2}{\vdash (Q_1; Q_2) : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2)}$$

The operator $\vee$ gives the largest of two security levels, that is $L \vee L = L$, $L \vee H = H$ and $H \vee H = H$. In the above rules, similar to the ones in [Smith, 2001; Boudol & Castellani, 2002], programs are assigned both a writing effect θ and a *testing effect* σ – it represents an upper-bound to the level of the variables that are tested in that program. This allows us to write the condition $\sigma_1 \leq \theta_2$ in the rule for sequential composition, implying that if high variables are tested in the first component, then low variables cannot be written to in the second

component. Notice that in the rules for conditionals and loops, the testing effect of the program is updated with that of the guard.

2.1.3 (Pre-)Lattices of Security Levels

So far we have considered the simple case where only two security levels are at hand – high and low. However, the above explanations can be easily extended to a more general setting, with an arbitrary number of security levels, and a flow relation for which certain operations "meet" and "join" can always be defined.

Lattices of Security Levels

Following the original works of Bell and La Padula and Denning [Bell & La Padula, 1976; Denning, 1976], it is standard to let security levels form a *lattice* (see for instance the survey [Sandhu, 1993] for the use of security lattices). We recall some basic definitions regarding partial order relations (reflexive, transitive and anti-symmetric).

Definition 2.1.1 (Least Upper-Bound and Greatest Lower-Bound)**.** *Given a partially ordered set* $(\mathcal{L}, \leq)$ *and two elements* l_1 *and* l_2 *of* $\mathcal{L}$*, we define:*

***Least Upper-Bound* of l_1, l_2** *is an element* l *that satisfies:*

$$l_1 \leq l$$
$$l_2 \leq l$$
$$l_1 \leq l_3 \text{ and } l_2 \leq l_3 \text{ implies } l \leq l_3$$

***Greatest Lower-Bound* of l_1, l_2** *is an element* l *that satisfies:*

$$l \leq l_1$$
$$l \leq l_2$$
$$l_3 \leq l_1 \text{ and } l_3 \leq l_2 \text{ implies } l_3 \leq l$$

Clearly, due to the anti-symmetry property of partial order relations, if the least upper-bound element or the greatest lower-bound element of a partial order exist, they are unique.

Definition 2.1.2 (Lattice)**.** *A partially ordered set* $(\mathcal{L}, \leq)$ *is a* lattice *if any two elements of* $\mathcal{L}$ *have a (unique) least upper-bound and a (unique) greatest lower-bound with respect to* $\leq$*.*

In a lattice we can define the *meet* ($\vee$) and *join* ($\wedge$) operation on elements of $\mathcal{L}$ such that, for any two $l_1, l_2 \in \mathcal{L}$ we have

$\boldsymbol{l_1 \wedge l_2}$ is the greatest lower-bound of l_1 and l_2, and

$\boldsymbol{l_1 \vee l_2}$ is the least upper-bound of l_1 and l_2.

Usually no further precisions are needed as to which kind of lattice is in use, and the general case is considered. Here we chose to deal with a specific kind of structure, which besides being convenient for the technical developments in the following chapters, is intuitive in practice. This structure forms a *pre-lattice* which, in spite of not being a lattice, still exhibits the necessary properties.

Pre-lattices of Security Levels

Even though the lattice structure is traditionally used for security levels, it is sufficient to use a preorder relation (reflexive, transitive but not necessarily anti-symmetric) as a flow relation. To obtain the corresponding structure, the *pre-lattice*, we extend the notion of *least upper-bound* and *greatest lower-bound* to preorder relations in the obvious way. Notice, however, that now these elements are not necessarily unique. We can then define:

Definition 2.1.3 (Pre-Lattice)**.** *A preordered set* $(\mathcal{L}, \preceq)$ *is a* pre-lattice *if any two elements of* $\mathcal{L}$ *have a least upper-bound and a greatest lower-bound with respect to* $\preceq$*.*

In a pre-lattice one can then define a *meet* ($\curlywedge$) and a *join* ($\curlyvee$) operation on elements of $\mathcal{L}$ that satisfy the same properties as the corresponding operations for lattices. To this end, one must choose, for each pair of elements l_1 and l_2, which of the greatest lower-bounds and least upper-bounds is given by the meet and join operators, respectively.

We leave the formal definition of the specific pre-lattices that we use in our framework, as well as the choice of meet and join operators that we adopt, to a later stage in this chapter (Section 2.3). Nevertheless, the basic ideas behind them stem from rather simple observations that we point out next.

Principals and Flow Policies

We are interested in the problem of making sure that the subjects of our system – the *principals* – can only read information that they are authorized to. For this reason, we define the security levels of objects to be the sets of principals that are authorized to read information contained in them. These are similar to access control lists. From this point of view, given a set $\boldsymbol{Pri}$ of principals, an object labeled $\boldsymbol{Pri}$ (also denoted $\bot$) is a most public one – every principal is allowed to read it –, whereas the label $\emptyset$ (also denoted $\top$) indi§¡cates a secret object, so secret that no principal is allowed to read it.

So far we have considered the case of a flat structure of principals, assuming no communication between them. However, once security levels have been assigned to objects, these can be used in programs running in different security environments, with different understandings of how information should be allowed to flow between principals. One might then wish to express that whatever principal p can read, also principal q can. This gives rise to the idea of a flow policy, which is a set of such statements. As an example, one could then say that the assignment

$$a_{\{p,q\}} := b_{\{p\}} \tag{2.9}$$

respects the flow policy $F = \{p \prec q\}$, while for instance the assignment

$$a_{\{p,q\}} := b_{\{q\}} \tag{2.10}$$

does not. Hence, in the ordering imposed by the flow policy F, the security level $\{q\}$ is lower than $\{p, q\}$. The concrete view of security levels as sets of principals, and of flow policies for specifying flow relations is adopted in the rest of this chapter.

Principals	$p, q \in \boldsymbol{Pri}$		
Security Levels	$l, j, k \subseteq \boldsymbol{Pri}$		
Effects	s	::=	$\langle l, l, l \rangle$
Type Variables	t		
Types	$\tau, \sigma, \theta \in \boldsymbol{Typ}$	::=	$t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta\ \mathrm{ref}_l \mid \tau \xrightarrow{s} \sigma$

Figure 2.1: Syntax of Security Annotations and Types

Variables	$x, y \in \boldsymbol{Var}$		
Reference Names	$a, b, c \in \boldsymbol{Ref}$		
Decorated Reference Names		::=	$a_{l,\theta}$
Values	$V \in \boldsymbol{Val}$	::=	$() \mid x \mid a_{l,\theta} \mid (\lambda x.M) \mid tt \mid ff$
Pseudo-values	$W \in \boldsymbol{Pse}$	::=	$V \mid (\varrho x.W)$
Expressions	$M, N \in \boldsymbol{Exp}$	::=	$W \mid (M\ N) \mid (M; N) \mid$ $(\mathrm{ref}_{l,\theta}\ M) \mid (!\ N) \mid (M := N) \mid$ $(\text{if } M \text{ then } N_t \text{ else } N_f) \mid$ $(\text{thread } M)$

Figure 2.2: Syntax of Expressions

Threads		::=	$M \quad (\in \boldsymbol{Exp})$
Pool of Threads	$P \in \mathbb{N}^{\boldsymbol{Exp}}$		
Store	S	:	$(\boldsymbol{Ref} \times 2^{\boldsymbol{Pri}} \times \boldsymbol{Typ}) \rightarrow \boldsymbol{Val}$
Configurations		::=	$\langle P, S \rangle$

Figure 2.3: Syntax of Configurations

2.2 An Imperative Concurrent λ-Calculus

We now define the language on which we base the study of the present chapter. We present the syntax of the expressions of the language, including its security annotations, and the format of configurations; we give the (small step) semantics for configurations, which is defined using evaluation contexts; we state some basic properties of the language. The definitions regarding the syntax of the language are all gathered on page 13, while the ones for the semantics can be found on page 16.

2.2.1 Syntax

The language of expressions is a call-by-value λ-calculus extended with the imperative constructs of ML, conditional branching and boolean values. We also introduce the possibility of dynamically creating concurrent threads. We now describe the syntax of security annotations, types, expressions and configura-

tions.

Security Annotations and Types

According to the intuitions given earlier in this chapter, *security levels* l, j, k are sets of *principals*, which are ranged over by $p, q \in \boldsymbol{Pri}$ (see Figure 2.1). They are apparent in the syntax as they are associated to references (and reference creators). The security level of a reference is to be understood as the set of principals that are allowed to read the information contained in that reference.

Types and effects are apparent in the syntax of the language. Their syntax is given in Figure 2.1, and will be explained in Section 2.4. These annotations do not play any role in the operational semantics, but are used for the purpose of proving type soundness.

Expressions

We assume given two disjoint countable sets $\boldsymbol{Var}$ and $\boldsymbol{Ref}$. *Variables* x are chosen from a set $\boldsymbol{Var}$. *Names* or *addresses* are given to references ($a, b, c \in \boldsymbol{Ref}$). Reference names can be created at runtime. We add annotations (subscripts) to references: they are decorated with their security level and the type of the values that they can hold. A decorated reference $a_{l,\theta}$ is a triple made of an address a, a type θ and a security level l. In the following we may omit subscripts whenever they are not relevant, following the convention that the same name has always the same subscript.

The syntax of expressions is defined in Figure 2.2. *Values*, ranged over by $V \in \boldsymbol{Val}$, are special expressions that cannot compute, and include: the command $()$ that does nothing; the function abstraction $(\lambda x.M)$ with body M and parameter x; the boolean values tt and ff. The construct $(\varrho x.W)$, which binds the occurrences of x in the pseudo-value W, is used to express recursive values – it recursively executes the result of applying $(\lambda x.W)$ to itself.

The set $\boldsymbol{Exp}$ of *expressions*, ranged over by M, N, includes: the application $(M\ N)$ that applies the function that results from computing M to the result of the computation of N; the conditional (if M then N_t else N_f) that executes N_t or N_f depending on whether the computation of M renders tt or ff[1]; the sequential composition $(M; N)$ that executes N after the execution of M has terminated; the dereferencing operation $(!\ M)$ that, after M has executed and returned a decorated reference name, returns the value that the reference points to; the assignment $(M := N)$ of the value returned by the computation of N to the decorated reference name returned by the computation of M; finally, the thread creator (thread M), that spawns the thread M, that is to be executed concurrently, and returns $()$.

Other useful commands can be derived from the above expressions. For example, we can write the let construct (let $x = N$ in M) as $((\lambda x.M)\ N)$. We can write recursive functions as $(\varrho f.(\lambda x.M))$, close to (let rec $f = (\lambda x.M)$ in f) written in an ML-like notation. We denote by loop the expression $(\varrho x.x)$. We may encode while loops in the following standard way:

$$(\text{while } M \text{ do } N) \stackrel{\text{def}}{=} ((\varrho y.(\lambda x.(\text{if } M \text{ then } (N; (y\ x)) \text{ else } x)))\ ()) \qquad (2.11)$$

[1]The notation N_t and N_f is used only for notational convenience to distinguish the branches, and does not have any further meaning.

Configurations

The evaluation relation is a transition relation between configurations of the form $\langle P, S\rangle$ where: P is a pool (multiset) of *threads* (expressions) that run concurrently[2]; and the *memory* or *store* S is a mapping from a finite set of *decorated reference names* to values.

2.2.2 Semantics

We now define the semantics of the language as a small step operational semantics on configurations. To this end, we give some useful notations and conventions. We then describe the transitions on configurations, that are based on evaluation contexts, and state some properties of the semantics.

Basic Sets and Functions

Given a configuration $\langle P, S\rangle$, we define $\mathrm{dom}(S)$ as the set of decorated reference names that are mapped by S. We say that a reference name a is fresh in S if it does not occur, with any subscript, in $\mathrm{dom}(S)$, that is if $b_{l,\theta} \in \mathrm{dom}(S)$ implies that $b \neq a$. We denote by $\mathrm{rn}(P)$ the set of decorated reference names that occur in the expressions of P (this notation is extended in the obvious way to expressions). We let $\mathrm{fv}(M)$ be the set of variables occurring free in M.

We restrict our attention to *well formed configurations* $\langle P, S\rangle$ satisfying the following condition for memories, values stored in memories, and thread names:

- $\mathrm{rn}(P) \subseteq \mathrm{dom}(S)$, and
- $a_{l,\theta} \in \mathrm{dom}(S)$ implies $\mathrm{rn}(S(a_{l,\theta})) \subseteq \mathrm{dom}(S)$, and
- all occurrences of a name in a configuration are decorated in the same way.

We denote by $\{x \mapsto W\}M$ the capture-avoiding substitution of W for the free occurrences of x in M. The operation of adding or updating the image of an object z to z' in a mapping Z is denoted $[z := z']Z$.

Evaluation Contexts

In order to define the evaluation order, it is convenient to write expressions using evaluation contexts. Intuitively, the expressions that are placed in such contexts are to be executed first. We write $\mathrm{E}[M]$ to denote an expression where the subexpression M is placed in the evaluation context E, obtained by replacing the occurrence of $[]$ in E by M. The evaluation contexts of the language define a call-by-value evaluation order (see Figure 2.4). Evaluation is *not* allowed under threads that have not yet been created.

Small Step Semantics

The transitions of our (small step) semantics are defined between configurations. The evaluation rules are defined in Figure 2.5. We omit the set-brackets for pools

[2]The notation $\mathbb{N}^{\boldsymbol{Exp}}$ denotes the power-multiset of the set $\boldsymbol{Exp}$, i.e., the set of all multisets of $\boldsymbol{Exp}$.

$$\begin{array}{lrl} \textit{Evaluation Contexts} & \mathrm{E}, \mathrm{F} & ::= \quad [] \mid (\mathrm{E}\ N) \mid (V\ \mathrm{E}) \mid (\mathrm{E}; N) \mid \\ & & \quad (\mathrm{ref}_{l,\theta}\ \mathrm{E}) \mid (!\ \mathrm{E}) \mid (\mathrm{E} := N) \mid (V := \mathrm{E}) \mid \\ & & \quad (\text{if E then } N_t \text{ else } N_f) \end{array}$$

Figure 2.4: Evaluation Contexts

$$\begin{array}{rcl} \langle \mathrm{E}[((\lambda x.M)\ V)], S\rangle & \xrightarrow{0} & \langle \mathrm{E}[\{x \mapsto V\}M], S\rangle \\ \langle \mathrm{E}[(\text{if } \mathit{tt} \text{ then } N_t \text{ else } N_f)], S\rangle & \xrightarrow{0} & \langle \mathrm{E}[N_t], S\rangle \\ \langle \mathrm{E}[(\text{if } \mathit{ff} \text{ then } N_t \text{ else } N_f)], S\rangle & \xrightarrow{0} & \langle \mathrm{E}[N_f], S\rangle \\ \langle \mathrm{E}[(V; N)], S\rangle & \xrightarrow{0} & \langle \mathrm{E}[N], S\rangle \\ \langle \mathrm{E}[(\varrho x.W)], S\rangle & \xrightarrow{0} & \langle \mathrm{E}[(\{x \mapsto (\varrho x.W)\}\ W)], S\rangle \\ \\ \langle \mathrm{E}[(!\ a_{l,\theta})], S\rangle & \xrightarrow{0} & \langle \mathrm{E}[V], S\rangle, \ \textit{where } S(a_{l,\theta}) = V \\ \langle \mathrm{E}[(a_{l,\theta} := V)], S\rangle & \xrightarrow{0} & \langle \mathrm{E}[()], [a_{l,\theta} := V]S\rangle \\ \langle \mathrm{E}[(\mathrm{ref}_{l,\theta}\ V)], S\rangle & \xrightarrow{0} & \langle \mathrm{E}[a_{l,\theta}], [a_{l,\theta} := V]S\rangle, \ \textit{if } a \textit{ fresh in } S \\ \langle \mathrm{E}[(\text{thread } N)], S\rangle & \xrightarrow{N} & \langle \mathrm{E}[()], S\rangle \end{array}$$

$$\frac{\langle \{M\}, S\rangle \xrightarrow{0} \langle \{M'\}, S'\rangle}{\langle \{M\}, S\rangle \to \langle \{M'\}, S'\rangle} \qquad \frac{\langle \{M\}, S\rangle \xrightarrow{N} \langle \{M'\}, S'\rangle}{\langle \{M\}, S\rangle \to \langle \{M', N\}, S'\rangle}$$

$$\frac{\langle P, S\rangle \to \langle P', S'\rangle \quad \langle P \cup Q, S\rangle \textit{ is well formed}}{\langle P \cup Q, S\rangle \to \langle P' \cup Q, S'\rangle}$$

Figure 2.5: Semantics

that are singletons. We start by defining the transitions of a single thread. These are decorated with the thread N that is possibly spawned during that transition, where $N = ()$ if no thread is created. The last three rules define the (unlabeled) transitions of pools of threads; they use the information contained in the label to add any spawned threads to the pool of threads.

The evaluation of the following expressions depends only on the expressions themselves: the application of a function with parameter x and body M to a value V returns the substitution of all free occurrences of x in M by V; a conditional with a test on a boolean value V and branches N_t and N_f returns N_t if the value is tt and N_f if the value is ff; the sequential composition of a value and an expression N renders N; the fix point of a pseudo-value W bound by x results in the substitution of the free occurrences of x in W by the expression itself.

The evaluation of some expressions might depend on and change the store: the creation of a reference of security level l and type θ containing the value V returns a reference with a name that does not occur so far in the store (say a), and adds the pair $((a, l, \theta), V)$ to the store; the dereferencing of a reference returns the value to which the store maps that reference; the assignment of a value V to a reference $a_{l,\theta}$ returns $()$ and updates the store by replacing any occurrence of a pair $((a, l, \theta), V')$ (where V' is the old value of a) by $((a, l, \theta), V)$.

By the last rule we can see that the execution of a pool of threads is compositional.

Properties of the Semantics

One can prove that the semantics preserves the conditions for well-formedness of configurations. Furthermore, a configuration where the pool of threads consists of just one expression has at most one transition, up to the choice of new names.

Next we show a simple but crucial property of the semantics, pinpointing the situations in which two computations of the same thread can split, that is can yield different threads. Apart from the situations in which two distinct fresh references are created, this can only occur if the expression is about to read a reference that is given different values by the memories in the starting configurations.

Lemma 2.2.1 (Splitting Computations)**.**
If $\langle M, S_1\rangle \xrightarrow{N_1} \langle M_1', S_1'\rangle$ *and* $\langle M, S_2\rangle \xrightarrow{N_2} \langle M_2', S_2'\rangle$ *with* $M_1' \neq M_2'$ *and* $\mathrm{dom}(S_2' - S_2) = \mathrm{dom}(S_1' - S_1)$*, then* $N_1 = () = N_2$ *and there exist* E *and* $a_{l,\theta}$ *such that* $M = \mathrm{E}[(!\ a_{l,\theta})]$*, and* $M_1' = \mathrm{E}[S_1(a_{l,\theta})]$*,* $M_2' = \mathrm{E}[S_2(a_{l,\theta})]$ *with* $S_1' = S_1$ *and* $S_2' = S_2$*.*

Proof. Note that the only rule where the state is used is that for $\mathrm{E}[(!\ a_{l,\theta})]$. A case analysis on the transition $\langle M, S_1\rangle \xrightarrow{N} \langle M_1', S_1'\rangle$. □

The condition $\mathrm{dom}(S_2' - S_2) = \mathrm{dom}(S_1' - S_1)$ allows us to ignore the differences in the programs M_1' and M_2' that might result from the non-deterministic choice of new reference names.

2.3 The Non-interference Policy

In this section we formally define non-interference, the security property that is studied in this chapter. We start by presenting the security pre-lattices in terms of a flow relation that is parameterized by the global flow policy; then we exhibit an indistinguishability relation on memories that is based on that flow relation; we then give a bisimulation definition of non-interference, using the small-step semantics defined in Section 2.2; finally, the security property is justified with some examples.

2.3.1 Principal-Based Security Pre-Lattices

We have mentioned that in our approach security levels j, k, l are sets of principals $p, q \in \boldsymbol{Pri}$ representing read-access rights to references. Our aim is to insure that information contained in a reference a_{l_1} (omitting the type annotation) does not leak to another reference b_{l_2} that gives a read access to an unauthorized principal p, i.e., such that $p \in l_2$ but $p \notin l_1$. We can interpret the reverse inclusion of security levels as indicating allowed flows of information: if $l_1 \supseteq l_2$ then the value of a a_{l_1} may be transferred to b_{l_2}, since the principals allowed to read this value from b were already allowed to read it from a. As a matter of fact, reverse inclusion forms a lattice structure over the set of security levels.

Remark 2.3.1. *Given a set $\boldsymbol{Pri}$ of principals, the pair $(2^{\boldsymbol{Pri}}, \supseteq)$ is a lattice, where the meet and join are set union and intersection, respectively.*

Now we shall see how the above lattice can be customized by means of relations on principals. A *flow policy* is a binary relation over $\boldsymbol{Pri}$. We let F, G range over such relations. A pair $(p, q) \in F$ 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*". We must point out here that, since we are dealing with confidentiality (and not integrity) a flow policy will only affect the reading capabilities of programs (and not their writing capabilities). A pair (p, q) that is a member of a flow policy will most often be written $p \prec q$. We denote, as usual, by F^* the reflexive and transitive closure of F.

We now introduce the *preorder on security levels* $\preceq_F$ that is determined by the flow policy F. For this purpose we use the notion of *F-upward closure* of a security level l, as follows:

$$l \uparrow_F = \{q \mid \exists p \in l.\ p\ F^*\ q\} \tag{2.12}$$

The F-upward closure of l contains all the principals that are allowed by the policy F to read the contents of a reference labeled l. We can now derive (as in [Myers & Liskov, 1998; Almeida Matos & Boudol, 2005]) a more permissive *flow relation*

$$l_1 \preceq_F l_2 \stackrel{\text{def}}{\Leftrightarrow} \forall q \in l_2\ .\ \exists p \in l_1\ .\ p\ F^*\ q \Leftrightarrow (l_1 \uparrow_F) \supseteq (l_2 \uparrow_F) \tag{2.13}$$

and use it to define the pre-lattice that is determined by a flow policy. Notice that $\preceq_F$ extends $\supseteq$ in the sense that $\preceq_F$ is larger than $\supseteq$ and that $\preceq_\emptyset = \supseteq$.

Definition 2.3.2 (Security Pre-Lattice)**.** *Given a set* ***Pri*** *of principals and a flow policy F in* ***Pri*** $\times$ ***Pri****, the pair $(2^{\boldsymbol{Pri}}, \preceq_F)$ is a pre-lattice, where* meet *($\curlywedge_F$) and* join *($\curlyvee_F$) are given respectively by the union and intersection of the F-upward closures:*

$$l_1 \curlywedge_F l_2 = l_1 \cup l_2 \qquad l_1 \curlyvee_F l_2 = (l_1 \uparrow_F) \cap (l_2 \uparrow_F) \tag{2.14}$$

We call these pre-lattices security pre-lattices.

In this chapter the flow policy that determines the security pre-lattice that is in use is called the *global flow policy*, since it is vaid for all the threads in the system.

2.3.2 A Bisimulation-Based Definition

We now define our security property in terms of the above defined flow relation $\preceq_G$, where G is the global flow policy.

Low-Equality

"*Low-equality*" is an informal designation of an equality relation that considers as indistinguishable memories that coincide in their "low part". The relation is defined between memories whose references are labeled with security levels. The low part of a memory is defined with respect to a security level l that is considered to be "low", and consequently for all levels lower than l (with respect to the flow relation $\preceq_G$ that is in use). Intuitively, two memories are said to be low-equal if they have the same low-domain, and if they give the same values to all references that are labeled with low security levels. Notice that both l and G are used as parameters.

Definition 2.3.3 (Low Part of a Memory)**.** *The* low part of a memory S *with respect to a flow policy G and a security level l is given by:*

$$S \restriction^{G,l} \stackrel{def}{=} \{(a_{k,\theta}, V) \mid (a_{k,\theta}, V) \in S \;\&\; k \preceq_G l\}$$

The low-equality of memories is thus defined:

Definition 2.3.4 (Low-Equality)**.** *The* low-equality between memories S_1 *and S_2 with respect to a flow policy G and a security level l is given by:*

$$S_1 =^{G,l} S_2 \stackrel{def}{\Leftrightarrow} S_1 \restriction^{G,l} = S_2 \restriction^{G,l}$$

This relation is transitive, reflexive and symmetric.

The Security Property

Bisimulations are often used to relate non-deterministic programs according to their behavior. They provide a natural way of formulating security properties in non-deterministic settings – see [Sabelfeld & Sands, 2000; Smith, 2001; Boudol & Castellani, 2002; Focardi & Gorrieri, 1995; Sabelfeld & Sands, 2005] for the use of bisimulations in stating security properties, and [Lowe, 2004] for a review of other semantic models for information flow. The idea is that by using a carefully

designed bisimulation we can express the requirement that two programs are to be related if they show the same behavior on the low part of two memories. Then, if a program is shown to be bisimilar to itself, one can conclude that the high part of the memory has not interfered with the low part, i.e., no security leak has occurred. A secure program would then be one that is related to itself by an appropriate bisimulation.

Our security property is based on a notion of bisimulation for sets of threads P with respect to a low security level and, since the notion of "being low" uses the flow relation $\preceq_G$, the global flow policy G appears here as a parameter as well. In the following we denote by $\twoheadrightarrow$ the reflexive closure of the transitions $\rightarrow$.

Definition 2.3.5 ((G,l)-bisimulation)**.** *A* (G,l)-bisimulation *is a symmetric relation* $\mathcal{R}$ *on multisets of threads such that:*

$$P_1 \ \mathcal{R} \ P_2 \text{ and } \langle P_1, S_1\rangle \rightarrow \langle P_1', S_1'\rangle \text{ and } S_1 =^{G,l} S_2$$

and $(*)$ *implies*

$$\exists P_2', S_2' \ . \ \langle P_2, S_2\rangle \twoheadrightarrow \langle P_2', S_2'\rangle \text{ and } S_1' =^{G,l} S_2' \text{ and } P_1' \ \mathcal{R} \ P_2'$$

where:
$(*)$ $\mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset$

The condition $\mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset$ guarantees that any reference that is eventually created by P_1 does not conflict with free names of P_2.

Remark 2.3.6.

- *For any* G *and* l *there exists a* (G,l)*-bisimulation, like for instance the set* $\boldsymbol{Val} \times \boldsymbol{Val}$ *of pairs of values.*
- *The union of a family of* (G,l)*-bisimulations is a* (G,l)*-bisimulation.*

Consequently, there is a largest (G,l)-bisimulation, which is the union of all (G,l)-bisimulations.

Notation 2.3.7. *The largest* (G,l)*-bisimulation is denoted* $\approx_{G,l}$.

One should observe that the relation $\approx_{G,l}$ is not reflexive. This is in fact the whole point of the security relation, since as we can see from the following definition, it should only be "reflexive" with respect to secure programs. For instance, the insecure expression $(v_B := (!\ u_A))$ is not bisimilar to itself if $A \not\preceq_G B$.

Definition 2.3.8 (Non-interference with respect to G)**.** *A pool of threads* P *satisfies the* Non-interference *policy (or is* secure *from the point of view of Non-interference) with respect to the global flow policy* G *if it satisfies* $P \approx_{G,l} P$ *for all security levels* l*. We then write* $P \in \mathcal{NI}(G)$.

Examples of Insecure Programs

In the following examples we assume two principals H and L, and a global flow policy G consisting of the pair $L \prec H$. We denote, as usual, references with security levels $\{H\}$ or $\{L\}$ simply by a_H or b_L, leaving out the type and the brackets.

The standard examples of direct leaks and of control leaks (see Subsection 2.1.1), which in our language are written

$$(a_L := (!\ b_H)) \tag{2.15}$$

$$(\text{if } (!\ a_H) \text{ then } (b_L := tt) \text{ else } (b_L := ff)) \tag{2.16}$$

do not satisfy Non-interference.

Using a bisimulation approach to security allows us to reject termination leaks, like for instance

$$((\text{if } (!\ a_H) \text{ then } () \text{ else } \mathsf{loop}); (b_L := tt)) \tag{2.17}$$

where writing at level L depends on reading at level H (see [Andrews & Reitman, 1980; Boudol & Castellani, 2002; Heintze & Riecke, 1998; Sabelfeld & Myers, 2003; Smith, 2001; Volpano & Smith, 1997] on using bisimulations to prevent this kind of leaks). This is because one of the branches that might result from the conditional $(\mathsf{loop}; (b_L := tt))$ cannot simulate the other one $((); (b_L := tt))$ in its change of the low reference b_l. Another example of a termination leak that arises in higher-order settings is

$$((!\ a_H)(); (b_L := tt)) \tag{2.18}$$

Indeed, the dereferencing of the high reference a_H can be seen as a "high test" that might result in a terminating or non-terminating branch – take for instance, $(\lambda y.(\lambda x.x))$ or $(\lambda y.\mathsf{loop})$ as two possible values of a_H in two low-equal memories. The application of these functions to $()$ unravels two expressions with different behaviors, analogously to the previous example. Similarly, there is a termination leak in

$$(((\lambda x.(x\ ()))\ (!\ a_H)); (b_L := tt)) \tag{2.19}$$

since in one step we obtain Example 2.18.

The kind of bisimulation we use has been described as "strong" in [Sabelfeld & Sands, 2000], since each time a transition is matched, we restart the bisimulation game by comparing the resulting pools of threads in the context of any new low-equal memories, rather than continuing with the resulting configurations. This allows us to detect an illegal flow in the program[3]

$$(\text{if } (!\ w_X) \text{ then } (\text{if } (!\ w_X) \text{ then } () \text{ else } (v_L := (!\ u_H))) \text{ else } ()) \tag{2.20}$$

which can be unraveled by other programs that execute concurrently in the pool of threads.

2.3.3 Properties of Secure Programs

Security is compatible with composition by set union:

Proposition 2.3.9 (Compositionality)**.**

$$P \in \mathcal{NI}(G) \textit{ and } Q \in \mathcal{NI}(G) \textit{ implies } (P \cup Q) \in \mathcal{NI}(G)$$

Proof. If $\mathcal{S}_1$ is a (G, l)-bisimulation such that $P\ \mathcal{S}_1\ P$ and $\mathcal{S}_2$ is a (G, l)-bisimulation such that $Q\ \mathcal{S}_2\ Q$, then $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ is a (G, l)-bisimulation such that $(P \cup Q)\ \mathcal{S}\ (P \cup Q)$. □

[3]This demanding definition for bisimulations seems also appropriate for dealing with declassification as well as for a mobile code scenario, where the shared memory of a system of threads can be modified by incoming code. We will see this in the following chapters.

Operationally High Threads

There is a class of threads that have the property of never performing any change in the low part of the memory. These are classified as being "high" according to their behavior:

Definition 2.3.10 (Operationally High Threads)**.** *A set $\mathcal{H}$ of threads is said to be a set of* operationally (G, l)-high threads *if the following holds for any $M \in \mathcal{H}$:*

$$\langle M, S\rangle \xrightarrow{N} \langle M', S'\rangle \text{ implies } S =^{G,l} S'$$
$$\text{and both } M', N \in \mathcal{H}$$

Even if a thread does contain low assignments or low reference creations, it can be considered *operationally high* if these commands are never reached in any execution, or if the assignments do not change the value of any reference.

Remark 2.3.11.

- *For any G and l there exists a set of operationally (G, l)-high threads, like for instance* ***Val****.*
- *The union of a family of sets of operationally (G, l)-high threads is a set of operationally (G, l)-high threads.*

Therefore, there exists the largest set of operationally (G, l)-high threads:

Notation 2.3.12. *The union of all sets of operationally (G, l)-high threads is denoted by $\mathcal{H}_{G,l}$.*

We say that a thread M is an *operationally (G, l)-high thread* if $M \in \mathcal{H}_{G,l}$.

Comparison with Basic Non-interference

Our Non-interference policy generalizes "basic" non-interference for sequential programs (see Subsection 2.1.1). To see this, let us first recall that the latter is based on the "big-step" semantics of programs, that is on the relation $\langle M, S\rangle \Rightarrow S'$ that a program M establishes from an initial state of the memory S to the final state S'. We say that M satisfies Basic Non-interference if $\langle M, S_1\rangle \Rightarrow S_1'$ and $\langle M, S_2\rangle \Rightarrow S_2'$, for S_1 and S_2 that differ only regarding confidential information, implies that also S_1' and S_2' are equal as regards public information. Making the global flow policy explicit in the same way we have done earlier, we have:

$$\langle M, S_1\rangle \Rightarrow S_1' \text{ and } \langle M, S_2\rangle \Rightarrow S_2' \text{ and } S_1 =^{G,l} S_2 \text{ implies } S_1' =^{G,l} S_2' \quad (2.21)$$

Let us denote by $\boldsymbol{DExp}$ the set of expressions that are written without using (thread) and (ref). We will show that the expressions in $\boldsymbol{DExp}$ satisfying Non-interference with respect to a given global flow policy G also satisfy Basic Non-interference with respect to G. Letting $\xrightarrow{*}$ denote the reflexive and transitive closure of $\rightarrow$, the big-step semantics for expressions in $\boldsymbol{DExp}$ can be defined as follows:

$$\langle M, S\rangle \Rightarrow S' \overset{\text{def}}{\Leftrightarrow} \exists V \in \boldsymbol{Val}.\ \langle M, S\rangle \xrightarrow{*} \langle V, S'\rangle \quad (2.22)$$

It is easy to see that the evaluation mechanism is deterministic for $M \in \boldsymbol{DExp}$, and that if $\langle M, S\rangle \Rightarrow S'$ then $\mathrm{dom}(S') = \mathrm{dom}(S)$. Now assume that $M \in$

$\boldsymbol{DExp} \cap \mathcal{NI}(G)$, $\langle M, S_1\rangle \xrightarrow{*} \langle V, S_1'\rangle$ and $\langle M, S_2\rangle \xrightarrow{*} \langle V', S_2'\rangle$ with $S_1 =^{G,l} S_2$. Then there exist M' and S_2'' such that $\langle M, S_2\rangle \xrightarrow{*} \langle M', S_2''\rangle$, $V \approx_{G,l} M'$ and $S_1' =^{G,l} S_2''$. Since M is deterministic, we have $\langle M', S_2''\rangle \xrightarrow{*} \langle V', S_2'\rangle$, and from $\langle V, S_1'\rangle$ there must be a sequence of transitions matching the move from $\langle M', S_2''\rangle$ to $\langle V', S_2'\rangle$. This sequence must be empty, and we then have $S_1' =^{G,l} S_2'$.

2.4 Typing Non-interference

In this section we present a type and effect system [Lucassen & Gifford, 1988] that only accepts programs that satisfy Non-interference. We start by defining the notation used to express the typing judgments and by explaining their meaning; we then comment on the typing conditions used in the typing rules, by giving examples of different kinds of security leaks – including higher-order leaks – that illustrate why each condition is necessary; finally, we conclude by giving some properties of the type system, including a Subject Reduction and Soundness theorems.

2.4.1 A Type and Effect System

The type and effect system that we present here selects secure threads by ensuring the compliance of all information flows to the flow relation that is parameterized with the global flow policy G. To achieve this, it constructively determines the *effects* of each expression, which contain information on the security levels of the references that the expression reads and writes, as well as the level of the references on which the termination of the computations might depend.

Typing Judgments

As defined in Figure 2.6, the judgments of the type and effect system have the form:

$$\Gamma \vdash_G M : s, \tau \tag{2.23}$$

The meaning of the parameters is the following:

- The typing context Γ assigns types to variables.
- The expression M is a program.
- The security effect s, of the form $\langle s.r, s.w, s.t\rangle$, can be understood as follows:
  - $s.r$ is the *reading effect*, an upper-bound on the security levels of the references that are read by M;
  - $s.w$ is the *writing effect*, a lower bound on the references that are written by M
  - $s.t$ is the *termination effect*, an upper bound on the level of the references on which the termination of expression M might depend.

  According to these intuitions, in the type system the reading and termination levels are composed in a covariant way, whereas the writing level is contravariant.

- The type τ is the type of the expression M. The syntax of types, which is given in Figure 2.1, is repeated here, for any type variable t:

$$\tau, \sigma, \theta \ \in \ \textbf{\textit{Typ}} \ ::= \ t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta \ \mathrm{ref}_l \mid \tau \xrightarrow{s} \sigma$$

  Typable expressions that reduce to () have type unit, and those that reduce to booleans have type bool. Typable expressions that reduce to a reference which points to values of type θ and has security level l have the reference type $\theta \ \mathrm{ref}_l$. The security level l is used to determine the effects of expressions that handle references. Expressions that reduce to a function that takes a parameter of type τ, that returns an expression of type σ, and with a *latent effect* s [Lucassen & Gifford, 1988] have the function type $\tau \xrightarrow{s} \sigma$. The latent effect is the security effect of the body of the function.

- The flow policy G is the global flow policy. As we have seen, it is used to parameterize the security pre-lattice, and in particular the flow relation and meet/join operators. The flow relation is used in the type system for imposing restrictions on the information flows that the typing rules allow, and the meet/join operators are used to construct the security effects of the expressions.

In some of the typing rules we use the join operation on security effects:

Definition 2.4.1.

$$s \curlyvee_G s' \quad \stackrel{def}{\Leftrightarrow} \quad (s.r \curlyvee_G s'.r, s.w \curlywedge_G s'.w, s.t \curlyvee_G s'.t)$$

The type and effect system is given in Figure 2.7. Notice that it is syntax directed, since there is exactly one rule per construction of the language. We use some abbreviations: we write the flow relation $\preceq$, meet $\curlywedge$ and join $\curlyvee$ with respect to the global flow policy, instead of $\preceq_G$, $\curlywedge_G$ and $\curlyvee_G$, respectively; we also omit the global flow policy that appears as subscript of $\vdash_G$ and simply write $\vdash$. Whenever we have $\Gamma \vdash M : \langle \bot, \top, \bot \rangle, \tau$, we only write $\Gamma \vdash M : \tau$.

2.4.2 Typing Conditions

We must now convince ourselves that the type system does indeed select only safe threads, according to the Non-interference policy, with respect to the security notion defined in the previous section. We give informal justifications to each side condition that constrains the typing of expressions and the construction of the security effects. We start by justifying the parts of the rules that reject programs that are insecure due to direct leaks and control leaks. We then look at the use of the termination effect for typing away termination leaks. The treatment of higher-order leaks is explained last.

Direct Leaks and Control Leaks

The reading and writing effects are respectively introduced by the constructs for dereferencing (see Der) and creating or updating the memory (see the typing rules Ref and Ass).

Typing Environments	Γ :	$\boldsymbol{Var} \to \boldsymbol{Typ}$
Global Flow Policies	$G \subseteq$	$\boldsymbol{Pri} \times \boldsymbol{Pri}$
Typing Judgments	::=	$\Gamma \vdash_G M : s, \tau$

Figure 2.6: Syntax of Typing Judgments (see also Figure 2.1)

$$[\textsc{Nil}]\ \Gamma \vdash () : \mathsf{unit}$$

$$[\textsc{Abs}]\ \frac{\Gamma, x : \tau \vdash M : s, \sigma}{\Gamma \vdash (\lambda x.M) : \tau \xrightarrow{s} \sigma} \qquad [\textsc{Rec}]\ \frac{\Gamma, x : \tau \vdash s, W : \tau}{\Gamma \vdash (\varrho x.W) : \tau}$$

$$[\textsc{BoolT}]\ \Gamma \vdash tt : \mathsf{bool} \qquad [\textsc{BoolF}]\ \Gamma \vdash ff : \mathsf{bool}$$

$$[\textsc{Var}]\ \Gamma, x : \tau \vdash x : \tau \qquad [\textsc{Loc}]\ \Gamma \vdash a_{l,\theta} : \theta\ \mathrm{ref}_l$$

$$[\textsc{Ref}]\ \frac{\Gamma \vdash M : s, \theta \quad s.r \preceq l}{\Gamma \vdash (\mathrm{ref}_{l,\theta}\ M) : s \curlyvee \langle \bot, l, \bot \rangle, \theta\ \mathrm{ref}_l}$$

$$[\textsc{Der}]\ \frac{\Gamma \vdash M : s, \theta\ \mathrm{ref}_l}{\Gamma \vdash (!\ M) : s \curlyvee \langle l, \top, \bot \rangle, \theta}$$

$$[\textsc{Ass}]\ \frac{\Gamma \vdash M : s, \theta\ \mathrm{ref}_l \quad \Gamma \vdash N : s', \theta \qquad \begin{array}{r} s.t \preceq s'.w \\ s.r, s'.r \preceq l \end{array}}{\Gamma \vdash (M := N) : s \curlyvee s' \curlyvee \langle \bot, l, \bot \rangle, \mathsf{unit}}$$

$$[\textsc{Cond}]\ \frac{\Gamma \vdash M : s, \mathsf{bool} \qquad \begin{array}{l} \Gamma \vdash N_t : s_t, \tau \\ \Gamma \vdash N_f : s_f, \tau \end{array} \qquad s.r \preceq s_t.w, s_f.w}{\Gamma \vdash (\text{if } M \text{ then } N_t \text{ else } N_f) : s \curlyvee s_t \curlyvee s_f \curlyvee \langle \bot, \top, s.r \rangle, \tau}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash M : s, \tau \xrightarrow{s'} \sigma \quad \Gamma \vdash N : s'', \tau \qquad \begin{array}{r} s.t \preceq s''.w \\ s.r, s''.r \preceq s'.w \end{array}}{\Gamma \vdash (M\ N) : s \curlyvee s' \curlyvee s'' \curlyvee \langle \bot, \top, s.r \curlyvee s''.r \rangle, \sigma}$$

$$[\textsc{Seq}]\ \frac{\Gamma \vdash M : s, \tau \quad \Gamma \vdash N : s', \sigma \quad s.t \preceq s'.w}{\Gamma \vdash (M; N) : s \curlyvee s', \sigma}$$

$$[\textsc{Thr}]\ \frac{\Gamma \vdash M : s, \mathsf{unit}}{\Gamma \vdash (\text{thread } M) : \langle \bot, s.w, \bot \rangle, \mathsf{unit}}$$

Figure 2.7: Type and Effect System

Cond. The constraint $s.r \preceq s_t.w, s_f.w$ insures that the branches N_t and N_f only assign to references with security level greater than the reading level of M. This prevents control leaks like the one in Example 2.16. Similarly, in order to reject the program

$$(\text{if } (!\ a_H) \text{ then } (\text{thread } (b_L := tt)) \text{ else } ()) \tag{2.24}$$

we require the writing level of M to be kept in the effect of (thread M).

Ass. The condition $s'.r \preceq l$ prevents direct flows, as in Example 2.15. Furthermore, the condition $s.r \preceq l$ rules out expressions like

$$((!\ a_H) := tt) \tag{2.25}$$

Indeed, the value of the reference a_H in different low-equal memories might be different references, with level L. In these cases, depending on the contents of a_H, different low assignments are performed.

App. The condition $s''.r \preceq s'.w$ prevents direct flows from the argument of the function via an assignment occurring in its body, like in

$$((\lambda x.(b_L := x))\ (!\ a_H)) \tag{2.26}$$

Ref. The condition $s.r \preceq l$ excludes flows like the one in the expression:

$$(\text{ref}_L\ (!\ u_H)) \tag{2.27}$$

This program creates a low reference that points to potentially different values, in case the result of the high dereference is different in the considered low-equal memories.

Termination Leaks

The termination effect is introduced in conditional (Cond) and application (App) constructs. In the conclusion of Cond, we add the reading level of the test to the termination level of the whole expression. This is because the conditional might choose branches with different termination-behavior depending on the references that it reads in the predicate. As to why the reading levels of both function and argument are recorded in the termination level of the application (App), consider Example 2.18 and 2.19, respectively. They show how the application of some argument to a dereferenced value can also unravel expressions with different termination behavior (thus depending on the reference that is read). Thread creation expressions (thread M) have no termination effect, since their evaluation always terminates in one step. Furthermore, since a spawned thread executes in parallel with its creating thread, the reference it reads and its termination behavior cannot influence future computations of the creating thread. Hence, its reading and termination effects are set to $\bot$.

Seq. The condition $s.t \preceq s'.w$ prevents termination leaks as in Example 2.17. Notice that this constraint is not as strict as "no low write after a high read", and allows us to accept for instance the secure program of Example 2.37.

Ass. The condition $s.t \preceq s'.w$ prevents termination leaks, similarly to the previous example, as in

$$((\text{if } (!\ a_H) \text{ then } b_H \text{ else } \mathsf{loop}) := (b_L := tt)) \tag{2.28}$$

App. the condition $s.t \preceq s''.w$ rules out the expression

$$(\text{if } (!\ a_H) \text{ then } \lambda x x \text{ else } \mathsf{loop})(b_L := tt) \tag{2.29}$$

in a way similar to the two previous examples.

There are some implicit conditions in the type system that prevent termination leaks. These are treated in the same way as the control leaks presented earlier, and do not need an explicit condition on the termination effect because the reading effect is never lower than the termination effect.

Cond. The constraint $s.r \preceq s_t.w, s_f.w$ also insures that the branches N_t and N_f only assign to references with security level greater than the termination level of M. This prevents termination leaks like the one in the following example:

$$(\text{if } (\text{if } (!\ a_H) \text{ then } tt \text{ else } \mathsf{loop}) \text{ then } (b_L := tt) \text{ else } (b_L := tt)) \tag{2.30}$$

Ass. The condition $s'.r \preceq l$ prevents termination leaks like:

$$(b_L := (\text{if } (!\ a_H) \text{ then } () \text{ else } \mathsf{loop})) \tag{2.31}$$

App. The condition $s''.r \preceq s'.w$ also rejects termination leaks from the argument of the function via an assignment occurring in its body, like in:

$$((\lambda x.(v_L := x))\ (\text{if } (!\ a_H) \text{ then } () \text{ else } \mathsf{loop})) \tag{2.32}$$

Ref. The condition $s.r \preceq l$ excludes termination flows like the one in the expression:

$$(\text{ref}_L\ (\text{if } (!\ a_H) \text{ then } () \text{ else } \mathsf{loop})) \tag{2.33}$$

Higher-Order Leaks

App. The condition $s.r \preceq s'.w$ excludes expressions that obtain from a high reference a function with a low latent write effect, and then unravel this low write effect by applying it to some argument. For instance, it rules out the following expression

$$((!\ a_{H,\theta})\ ()) \tag{2.34}$$

where $\theta = \mathsf{unit} \xrightarrow{(\bot,L,\bot)} \mathsf{unit}$. This program, similar to Example 2.6, turns out to be dangerous for instance if the value of the reference a is $(\lambda z.(b_{L,\theta'} := V))$, with different values for V in different memories.

Derived Typing Rules

We can derive the typing of (let $x = N$ in M), using the typing rules ABS and APP:

$$\frac{\Gamma \vdash N : s, \tau \quad \Gamma, x : \tau \vdash M : s', \sigma \quad s.r \preceq s'.w}{\Gamma \vdash (\text{let } x = N \text{ in } M) : s \curlyvee s' \curlyvee (\bot, \top, s.r), \sigma} \tag{2.35}$$

The derived typing for the while construct, using recursion is:

$$\frac{\Gamma \vdash M : s, \mathsf{bool} \quad \Gamma \vdash N : s', \tau \quad s.r, s'.t \preceq s.w, s'.w}{\Gamma \vdash (\text{while } M \text{ do } N) : s \curlyvee s' \curlyvee (\bot, \top, s.r), \mathsf{unit}} \tag{2.36}$$

In the resulting rule, the security level of the guard expression is recorded as part of the termination level of the while construct, as in [Boudol & Castellani, 2002; Smith, 2001].

We could have used an encoding of $(M; N)$ by (let $z = M$ in N), where $z \notin \text{fv}(N)$ and used a derived typing rule for it. However, due to the update of the termination effect with the reading effect of N, it would be slightly more restrictive than SEQ. For instance neither (let $x = (w_H := (!\ u_H))$ in $(v_L := tt)$) nor $((\lambda x(w_H := x)((!\ u_H)))); (v_L := tt))$ can be typed (in our type system), whereas the expression

$$((w_H := (!\ u_H)); (v_L := tt)) \tag{2.37}$$

is accepted. Similarly, the derived typing rules for $(\text{ref}_{l,\theta}\ M)$ with respect to $(\lambda x.(\text{ref}_{l,\theta}\ x))M$, for $(!\ M)$ with respect to $((\lambda x.(!\ x))\ M)$, and for $(M := N)$ with respect to $(((\lambda x.(\lambda y.(x := y)))\ M)\ N)$ are more restrictive than the typings that are given by the rules REF, DER and ASS, respectively.

We have justified all the constraints on information flow that appear in the typing rules. The completeness of this informal analysis will be established by the type soundness result.

2.4.3 Properties of Typed Expressions

The same intermediate results are stated in the next two chapters, in settings that are increasingly complex, but with analogous proofs. For this reason we omit the details here and refer the reader to Subsection 4.4.3 for a complete proof.

Subject Reduction

The first main result is Subject Reduction, which states that the type of a thread is preserved by reduction. When a thread performs a computation step, some of its effects may be performed by reading, updating or creating a reference, and some may be discarded when a branch in a conditional expression is taken. Then the effects of an expression "weaken" along the computations. To prove it, we assume that the value contained in references of type θ, in the memories that we are dealing with, have indeed type θ.

Theorem 2.4.2 (Subject Reduction)**.**
If for some s, τ we have $\Gamma \vdash M : s, \tau$ and $\langle M, S\rangle \xrightarrow{N} \langle M', S'\rangle$ where all $a_{l,\theta} \in \text{dom}(S)$ satisfy $\Gamma \vdash S(a_{l,\theta}) : \theta$, then $\exists s'$ such that $\Gamma \vdash M' : s', \tau$ and

$s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$. Furthermore, $\exists s''$ such that $\Gamma \vdash N : s'', \mathsf{unit}$ and $s.w \preceq s''.w$.

Proof. The main proof is a case analysis on the transition $\langle M, S\rangle \xrightarrow{N} \langle M', S'\rangle$. See the detailed proof of Theorem 4.4.7 and preceding lemmas. □

Syntactically High Expressions

Some expressions can be easily classified as "high" by the type system, which only considers their syntax. These cannot perform changes to the "low" memory simply because their code does not contain any instruction that could perform them. Since the writing effect is intended to be a lower bound to the level of the references that the expression can create or assign to, expressions with a high writing effect can be said to be *syntactically high*:

Definition 2.4.3 (Syntactically High Expressions)**.** *An expression M is* syntactically (G, l)-high *if there exists Γ, s, τ such that $\Gamma \vdash_G M : s, \tau$ with $s.w \not\preceq_G l$. The expression M is a* syntactically (G, l)-high function *if there exists Γ, s, τ such that $\Gamma \vdash_G M : \tau \xrightarrow{s} \sigma$ with $s.w \not\preceq_G l$.*

We can now state that syntactically high expressions have an operationally high behavior.

Lemma 2.4.4 (High Expressions)**.** *If M is a syntactically (G, l)-high expression, then M is an operationally (G, l)-high thread.*

Proof. See proof of Lemma 4.4.9. □

2.4.4 Soundness

The final result of this chapter, soundness, states that the type system only accepts expressions that are secure in the sense of Definition 2.3.8, for a global flow policy G. In the remainder of this section we sketch the main definitions and results that can be used to reconstruct a direct proof of this result. A similar proof is given in detail for the richer language of Chapter 4.

We first build a symmetric binary relation between typable expressions whose terminating behaviors do not depend on high references, more precisely, between those that are typable with a low termination effect. The binary relation should be such that if the evaluation of two related expressions, in the context of two low-equal memories should split (see Lemma 2.2.1), then the resulting expressions are still in the relation. This relation, called $\mathcal{T}_{G,low}$, is inductively defined for a security level *low* in Figure 2.8.

The next proposition states that $\mathcal{T}_{G,low}$ is a kind of "strong bisimulation" with respect to the transition relation $\xrightarrow{N}$.

Proposition 2.4.6 (Strong Bisimulation for Low-Terminating Threads)**.** *If we have $M_1\ \mathcal{T}_{G,low}\ M_2$ and $\langle M_1, S_1\rangle \xrightarrow{N} \langle M_1', S_1'\rangle$, with $S_1 =^{G,low} S_2$ such that a is fresh for S_2 if $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$, then there exist M_2' and S_2' such that $\langle M_2, S_2\rangle \xrightarrow{N} \langle M_2', S_2'\rangle$ with $M_1'\ \mathcal{T}_{G,low}\ M_2'$, $S_1' =^{G,low} S_2'$ and $\mathrm{dom}(S_1' - S_1) = \mathrm{dom}(S_2' - S_2)$.*

Proof. By induction on the definition of $\mathcal{T}_{G,low}$. □

Definition 2.4.5 ($\mathcal{T}_{G,low}$)**.** *We have that* $M_1\ \mathcal{T}_{G,low}\ M_2$ *if* $\Gamma \vdash_G M_1 : s_1, \tau$ *and* $\Gamma \vdash_G M_2 : s_2, \tau$ *for some* Γ*,* s_1*,* s_2 *and* τ *with* $s_1.t \preceq_G low$ *and* $s_2.t \preceq_G low$ *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* $M_2 = (\bar{M}_2; \bar{N})$ *where* $\bar{M}_1\ \mathcal{T}_{G,low}\ \bar{M}_2$*, or*

Clause 4. $M_1 = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ *and* $M_2 = (\mathrm{ref}_{l,\theta}\ \bar{M}_2)$ *where* $\bar{M}_1\ \mathcal{T}_{G,low}\ \bar{M}_2$*, and* $l \not\preceq_G low$*, or*

Clause 5. $M_1 = (!\ \bar{M}_1)$ *and* $M_2 = (!\ \bar{M}_2)$ *where* $\bar{M}_1\ \mathcal{T}_{G,low}\ \bar{M}_2$*, or*

Clause 6. $M_1 = (\bar{M}_1 := \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 := \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{T}_{G,low}\ \bar{M}_2$*, and* $\bar{N}_1\ \mathcal{T}_{G,low}\ \bar{N}_2$*, and* $\bar{M}_1, \bar{M}_2$ *both have type* $\theta\ \mathrm{ref}_l$ *for some* θ *and* l *such that* $l \not\preceq_G low$*.*

Figure 2.8: The relation $\mathcal{T}_{G,low}$

We now define a larger symmetric binary relation on typable expressions. Similarly to the previous one, it should be possible to relate the results of the computations of two related expressions in the context of two low-equal memories. The binary relation $\mathcal{R}_{G,low}$ on expressions is defined inductively in Figure 2.9. The relation $\mathcal{R}_{G,low}$ is a kind of "strong bisimulation", with respect to the transition relation $\xrightarrow{N}$:

Proposition 2.4.8 (Strong Bisimulation for Low Typable Threads)**.** *Suppose that* $M_1\ \mathcal{R}_{G,low}\ M_2$ *and that* $M_1 \notin \mathcal{H}_{G,low}$*. If* $\langle M_1, S_1\rangle \xrightarrow{N} \langle M_1', S_1'\rangle$*, with* $S_1 =^{G,low} S_2$ *such that* a *is fresh for* S_2 *if* $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$*, then there exist* M_2' *and* S_2' *such that* $\langle M_2, S_2\rangle \xrightarrow{N} \langle M_2', S_2'\rangle$ *with* $M_1'\ \mathcal{R}_{G,low}\ M_2'$*,* $S_1' =^{G,low} S_2'$*, and* $\mathrm{dom}(S_1' - S_1) = \mathrm{dom}(S_2' - S_2)$*.*

Proof. By induction on the definition of $\mathcal{R}_{G,low}$. □

To conclude the proof of the Soundness Theorem, we must exhibit a bisimulation on pools of threads. Consider the following relation:

Definition 2.4.9 ($\mathcal{R}^\star_{G,low}$)**.** *The relation* $\mathcal{R}^\star_{low}$ *is inductively defined as follows:*

$$a)\ \frac{M \in \mathcal{H}_{G,low}}{\{M\}\ \mathcal{R}^\star_{G,low}\ \emptyset} \qquad b)\ \frac{M \in \mathcal{H}_{G,low}}{\emptyset\ \mathcal{R}^\star_{G,low}\ \{M\}} \qquad c)\ \frac{M_1\ \mathcal{R}_{G,low}\ M_2}{\{M_1\}\ \mathcal{R}^\star_{G,low}\ \{M_2\}}$$

$$d)\ \frac{P_1\ \mathcal{R}^\star_{G,low}\ P_2 \quad Q_1\ \mathcal{R}^\star_{G,low}\ Q_2}{(P_1 \cup Q_1)\ \mathcal{R}^\star_{G,low}\ (P_2 \cup Q_2)}$$

We will now use Strong Bisimulation for Low Typable Threads (Proposition 2.4.8) to prove the following:

Proposition 2.4.10. *The relation* $\mathcal{R}^\star_{G,low}$ *is a* (G, low)*-bisimulation.*

Definition 2.4.7 ($\mathcal{R}_{G,low}$)**.** *We have that* $M_1\ \mathcal{R}_{G,low}\ M_2$ *if* $\Gamma \vdash_G M_1 : s_1, \tau$ *and* $\Gamma \vdash_G M_2 : s_2, \tau$ *for some* Γ*,* s_1*,* s_2 *and* τ *and one of the following holds:*

Clause 1'. $M_1, M_2 \in \mathcal{H}_{G,low}$*, or*

Clause 2'. $M_1 = M_2$*, or*

Clause 3'. $M_1 = (\text{if } \bar{M}_1 \text{ then } \bar{N}_t \text{ else } \bar{N}_f)$ *and* $M_2 = (\text{if } \bar{M}_2 \text{ then } \bar{N}_t \text{ else } \bar{N}_f)$ *with* $\bar{M}_1\ \mathcal{R}_{G,low}\ \bar{M}_2$ *and* $\bar{N}_t, \bar{N}_f \in \mathcal{H}_{G,low}$*, or*

Clause 4'. $M_1 = (\bar{M}_1\ \bar{N}_1)$ *and* $M_2 = (\bar{M}_2\ \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{R}_{G,low}\ \bar{M}_2$*, and* $\bar{N}_1, \bar{N}_2 \in \mathcal{H}_{G,low}$*, and* $\bar{M}_1, \bar{M}_2$ *are syntactically* (G, low)*-high functions, or*

Clause 5'. $M_1 = (\bar{M}_1\ \bar{N}_1)$ *and* $M_2 = (\bar{M}_2\ \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{T}_{low}\ \bar{M}_2$*, and* $\bar{N}_1\ \mathcal{R}_{G,low}\ \bar{N}_2$*, and* $\bar{M}_1, \bar{M}_2$ *are syntactically* (G, low)*-high functions, or*

Clause 6'. $M_1 = (\bar{M}_1; \bar{N})$ *and* $M_2 = (\bar{M}_2; \bar{N})$ *with* $\bar{M}_1\ \mathcal{R}_{G,low}\ \bar{M}_2$ *and* $\bar{N} \in \mathcal{H}_{G,low}$*, or*

Clause 7'. $M_1 = (\bar{M}_1; \bar{N})$ *and* $M_2 = (\bar{M}_2; \bar{N})$ *with* $\bar{M}_1\ \mathcal{T}_{G,low}\ \bar{M}_2$*, or*

Clause 8'. $M_1 = (\text{ref}_{l,\theta}\ \bar{M}_1)$ *and* $M_2 = (\text{ref}_{l,\theta}\ \bar{M}_2)$ *where* $\bar{M}_1\ \mathcal{R}_{G,low}\ \bar{M}_2$*, and* $l \not\preceq_G low$*, or*

Clause 9'. $M_1 = (!\ \bar{M}_1)$ *and* $M_2 = (!\ \bar{M}_2)$ *where* $\bar{M}_1\ \mathcal{R}_{G,low}\ \bar{M}_2$*, or*

Clause 10'. $M_1 = (\bar{M}_1 := \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 := \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{R}_{G,low}\ \bar{M}_2$*, and* $\bar{N}_1, \bar{N}_2 \in \mathcal{H}_{G,low}$*, and* $\bar{M}_1, \bar{M}_2$ *both have type* $\theta\ \text{ref}_l$ *for some* θ *and* l *such that* $l \not\preceq_G low$*, or*

Clause 11'. $M_1 = (\bar{M}_1 := \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 := \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{T}_{G,low}\ \bar{M}_2$*, and* $\bar{N}_1\ \mathcal{R}_{G,low}\ \bar{N}_2$*, and* $\bar{M}_1, \bar{M}_2$ *both have type* $\theta\ \text{ref}_l$ *for some* θ *and* l *such that* $l \not\preceq_G low$*.*

Figure 2.9: The relation $\mathcal{R}_{G,low}$

We now state the main result of the chapter, saying that our type system only accepts threads that can securely run concurrently with other typable threads.

Theorem 2.4.11 (Soundness for Non-interference)**.** *Consider a pool of threads P and a global flow policy G. If for any $M \in P$ there exist Γ, s, and τ such that $\Gamma \vdash_G M : s, \tau$, then P satisfies the Non-interference policy with respect to G.*

Proof. By Clause 2' of Definition 2.4.7, for all choices of security levels *low*, we have that $M \; \mathcal{R}_{G,low} \; M$. By Rule c) of Definition 2.4.9, $\{M\} \; \mathcal{R}^{\star}_{G,low} \; \{M\}$. Since this is true for all $M \in P$, by Rule d) we have that $P \; \mathcal{R}^{\star}_{G,low} \; P$. By Proposition 2.4.10 we conclude that $P \approx_{G,low} P$. □

The above result is compositional, in the sense that it is sufficient to verify the typability of each thread separately in order to ensure non-interference for the whole pool of threads.

2.5 Related Work

2.5.1 Types and Effects

Most studies concerning security for functional languages, regard values as having a security level. While this is inevitable in pure functional languages, this viewpoint has been adopted even where imperative features are considered, for instance in [Heintze & Riecke, 1998; Pottier & Simonet, 2003; Zdancewic & Myers, 2002]. However, one might wonder what it means for 4242 to be secret *per se*. It could become secret when used for example as a PIN code, but in that case it is the PIN code in itself that is secret, whatever the value it might take.

Our standpoint is that security levels are associated with the objects in which information is stored or communicated, like files, libraries, databases, channels or references (as in our language). Then, it is the accesses (typically read or write) that are to be controlled. This scenario is a natural candidate for the use of a *type and effect system* [Lucassen & Gifford, 1988], where read and write accesses are seen as producing security effects that are controlled with flow constraints. In our approach, effects are security levels that are built from those that are attached to references (as noted in [Crary *et al.*, 2005], the security levels of references play the role of *regions*). Consequently, all pure expressions (written without $(!\ N)$ and $(M := N)$), including values, are secure.

This view is consistent with most studies dealing with imperative languages. As a matter of fact, our type system appears to be a quite direct generalization of systems like [Volpano *et al.*, 1996; Smith, 2001; Boudol & Castellani, 2002]. In the work [Volpano *et al.*, 1996] and many others that followed, "expressions" are either assigned to values or tested in boolean predicates, and are assumed not to create or update references, and not to have the potential to diverge. Then the security effect of expressions is reduced to its reading level r – in our setting, it would have security effect $\langle r, \top, \bot \rangle$. On the other hand, the programs, or "commands" in [Volpano *et al.*, 1996], only read the store when evaluating an expression, and therefore it is enough to record their writing level w in $\langle \bot, w, \bot \rangle$. When termination leaks are considered, the termination level t (the "guard level" of [Boudol & Castellani, 2002] or the "running time level"

of [Smith, 2001]) is added to the security effect of commands, thus obtaining $\langle \bot, w, t \rangle$.

A store-oriented view of confidentiality is also followed in the type system of [Crary *et al.*, 2005], but with an important difference: expressions are assumed to never write below their read level. Then, the secure program of Example 2.37 is accepted thanks to the "informativeness" predicate, which doesn't seem to have a counter-part here. Another difference is that [Crary *et al.*, 2005] does not consider termination leaks. This is the purpose of our "termination effect".

2.5.2 Treatment of Termination Leaks

Treatment of termination leaks is beyond the realm of the classical approach to "basic" non-interference, founded on a big-step semantics, which actually corresponds to a variant of our type system where the termination effect is always $\bot$. When bisimulation-based approaches are considered, termination leaks can easily be ruled out for instance by allowing only predicates of the lowest security level to appear in while loops (see [Volpano & Smith, 1997; Smith & Volpano, 1998] for instance). However, this is obviously a very severe restriction. The counter-part in our language would be to restrict the reading effect of the conditional predicate to $\bot$. This is the advantage of using some form of reading effect in the type system.

The third component in the security effect – distinct from the reading effect – is a further refinement that addresses the fact that all reads performed by expressions must be recorded (by the reading effect) even though they do not always influence the termination behaviors. In fact, we would like to accept the expression of Example 2.37, but reject Example 2.34. This would not be possible if we had approximated $s.t$ as $s.r$.

Our type system could be further improved, by being more refined when extracting the termination level of an expression. For instance, we would like to accept the program

$$(\text{if } (!\, u_H) \text{ then } (v_H := tt) \text{ else } (v_H := ff)); (w_L := tt) \tag{2.38}$$

and other similar ones, where it is straightforward to anticipate that both branches of the conditional terminate. This was done in [Boudol, 2005b], where the termination level of the above conditional is taken as $\bot$.

Chapter 3

Non-disclosure and Declassification

In this chapter we address the issue 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. To this end we present *non-disclosure* as a property that generalizes non-interference and allows for declassification, and we demonstrate the use of a type and effect system for enforcing that policy. This dynamic view of information flow policies is supported by the principal-based security pre-lattice that was presented in the beginning of Section 2.3. We study the forms of information flow that are enabled by the non-disclosure policy, but are forbidden by the non-interference policy.

The chapter is organized as follows. In the next section we motivate the need for expressing declassification, and introduce the flow declaration as a tool that can be used under the control of a non-disclosure property. In Section 3.2 we present an extension of the imperative higher-order language of Section 2.2, to which we add a flow declaration construct. Then, in Section 3.3, we introduce our generalization of non-interference, namely the Non-disclosure policy, that takes into account dynamic flow policies. A type and effect system is given for the language in Section 3.4, as well as some basic properties of the type system, including soundness. We then discuss related work.

3.1 Introduction

3.1.1 Limitations of Non-interference

The applicability of non-interference has been a matter of debate for a long time. One of its problems is that, *by definition*, it rejects programs that deliberately *declassify* information from high security levels to lower ones, thus disabling the use of programs that are very common and even unavoidable.

A typical example, that was pointed out in [Jones & Lipton, 1975], is the password checking procedure, whose purpose is to restrict the access to some service to users that are in possession of a secret password. This could roughly

be written as:

$$\text{if } (input_L = password_H) \text{ then } \ldots \text{ else } print_L(\text{“Password is wrong.”}) \quad (3.1)$$

These programs reveal, to any user that happens to attempt to “log-in”, at least one bit of information when rejecting an inputted string. Another example is the one that delivers a password to users who have paid for the service it gives access to. This procedure might roughly look like this:

$$\text{if } paid \text{ then } print_L(password_H) \text{ else } \ldots \quad (3.2)$$

Clearly, such programs are considered insecure if the pre-lattice of security levels (see Section 2.1) disallows flows from L to H.

The above examples contain operations that declassify information regarding secret passwords. They would therefore not be allowed to run in a system whose policy requires programs to satisfy non-interference. However, one might want to accept such programs, while still being able to control the information flows that occur in other parts of the program (e.g. the first program, after the user has logged-in).

3.1.2 A View of Declassification

The search for mechanisms for allowing information release to occur under the scrutiny of some information flow control policy is a challenging problem that has motivated a lot of work. We will comment on this at the end of this chapter. However, as was observed in [Sabelfeld & Sands, 2005], various approaches to this issue aim at different goals.

Consider for instance a password checking procedure that returns the root password instead of a “Yes/No” answer. This program can be considered “wrong” from a semantical point of view, for releasing information that most probably was not initially intended. However, the technical challenge of providing downgrading facilities for having it intentionally accepted is no different from that for accepting Example 3.1. There are broadly two main approaches that can be considered:

1. How may we *justify* that a program is allowed to declassify information, i.e. that it does not actually reveal “too much”?

   Approaches to this question, which seem beyond the reach of static analysis techniques, aim at ensuring that it is not feasible to exploit the allowed information leakages to obtain “unintended” declassification. Techniques for achieving this include quantifications of the amount of information that a program may leak, or the use of complexity-theoretic or probabilistic arguments (see [Clark *et al.*, 2004; Di Pierro *et al.*, 2002; Laud, 2001; Laud, 2003; Volpano, 2000; Volpano & Smith, 2000], to mention just a few recent papers).

2. How may we *accept* such programs in a language-based security setting, while still preserving some secure information flow property?

   This question includes

- The language design issue of conceiving the appropriate tools to provide a programmer with means to express the intention of declassifying information.
- The semantical formalization issue of expressing the appropriate information flow property that secure programs should satisfy.
- The security enforcement issue of rejecting programs that do not satisfy the security property.

Here we tackle the second question, therefore leaving to the programmer the responsibility for writing a program that meets his own specification, regarding "what" and "how much" information he intends to release. Nevertheless, we keep in mind that it would be useful for the programmer to have means to check that his code implements only the intended information flows.

3.1.3 Flow Declaration

Given that deliberately downgrading programs are to be validated by the programmer, the programming language should be as flexible as possible in expressing them. To this end, we introduce in a core language a programming construct for dynamically manipulating the pre-lattice that establishes the legal information flows between security levels.

Recall (from Section 2.1) that the pre-lattices that we adopt here are derived from *flow policies*, which are binary relations that indicate how information may flow between the *principals* (or subjects) of the system. It is in fact these flow policies that are directly manipulated by our new programming construct, which we call *flow declaration*. The construct is written (flow F in M), and takes two parameters: a flow policy F, declaring that the flows expressed in F are valid; a program M, which is the scope of the flow declaration. The meaning is that M should comply to the flow policy that holds in the context where (flow F in M) is executed, *extended with* F.

As an example, if we have principals A and B, then – using the notation of the language of the previous chapter, leaving out the type and set brackets in the security level – the program

$$(\text{flow } A \prec B \text{ in } (b_B := (!\ a_A))) \tag{3.3}$$

is legal, since the declaration $A \prec B$ states that information is allowed to flow from A to B in the subprogram $(b_B := (!\ a_A))$. This is a declassification operation – unless, of course, flows from A to B are already allowed by the context where this program is placed. It should be clear, on the other hand, that a statement like

$$(\text{flow } C \prec B \text{ in } (b_B := (!\ a_A))) \tag{3.4}$$

is not legal, unless the current policy allows information to flow from level A to C (or B).

Making use of the local nature of the flow declarations, one can write:

$$((\text{flow } A \prec B \text{ in } M); (\text{flow } B \prec C \text{ in } N)) \tag{3.5}$$

which shows a way of achieving a kind of non-transitive flow relation (see [Rushby, 1992; Roscoe & Goldsmith, 1999; Bossi *et al.*, 2004]).

Using '||' to express concurrency, one can also have different flow policies ruling simultaneously in different portions of a program – like in:

$$(\text{flow } A \prec B \text{ in } M) \;||\; (\text{flow } C \prec D \text{ in } N) \tag{3.6}$$

3.1.4 From Non-interference to Non-disclosure

As was pointed out in [Ryan *et al.*, 2001], in spite of its rigidity, non-interference is still a simple and elegant concept that the security community would like to retain. It is therefore desirable to find an alternative to non-interference that would preserve its "spirit".

Our new confidentiality property, that we call *non-disclosure*, is designed to support declassification. It roughly says that a program P is secure if at each step it satisfies non-interference *with respect to the flow policy that holds for this step.*

We have seen in the previous chapter (in Section 2.3) that, particularly for concurrent settings, non-interference can be conveniently expressed using bisimulations. These are based on small-step semantics, that specify transitions between successive states of the program and the memory:

$$\langle P, S\rangle \rightarrow \langle P', S'\rangle \tag{3.7}$$

Expressing execution by means of small steps, as opposed to using big steps for describing the final result of a computation, is suitable for treating our flow declaration construct, where the safety of an execution step is defined locally. It suffices to label each transition with the flow policy F that is declared by the evaluation context

$$\langle P, S\rangle \underset{F}{\longrightarrow} \langle P', S'\rangle \tag{3.8}$$

in order to have at hand the information about which pre-lattice holds for that particular step. Hence, our non-disclosure policy will also be formulated in terms of a bisimulation, similar to the one that is used for non-interference. The intuition is that, as regards information flow, the memory S should be considered from the point of view of the current flow policy extended with F.

3.2 Adding a Flow Declaration Construct

We present now the language on which we base the study of the present chapter, by extending the one that is defined in the previous chapter (here we call it the *core language*) with a programming construct for directly manipulating flow policies – namely the *flow declaration* construct. We thus obtain an imperative higher-order λ-calculus with thread and reference creation and declassification. We only comment on the new features of the language, and refer the reader to Section 2.2 for further explanations. However, we give the full definitions of the syntax of the language, the (small step) semantics for configurations that is now decorated with the flow policies declared by the evaluation contexts, and also some basic properties of the language. The definitions regarding its syntax are all gathered in page 39, while the ones for the semantics can be found on page 41.

Principals	$p, q \in \boldsymbol{Pri}$		
Security Levels	$l, j, k \subseteq \boldsymbol{Pri}$		
Flow Policies	$F, G \subseteq \boldsymbol{Pri} \times \boldsymbol{Pri}$		
Effects	s	$::=$	$\langle l, l, l \rangle$
Type Variables	t		
Types	$\tau, \sigma, \theta \in \boldsymbol{Typ}$	$::=$	$t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta\ \mathrm{ref}_l \mid \tau \xrightarrow[F]{s} \sigma$

Figure 3.1: Syntax of Security Annotations and Types

Variables	$x, y \in \boldsymbol{Var}$		
Reference Names	$a, b, c \in \boldsymbol{Ref}$		
Decorated Reference Names		$::=$	$a_{l,\theta}$
Values	$V \in \boldsymbol{Val}$	$::=$	$() \mid x \mid a_{l,\theta} \mid (\lambda x.M) \mid tt \mid ff$
Pseudo-values	$W \in \boldsymbol{Pse}$	$::=$	$V \mid (\varrho x.W)$
Expressions	$M, N \in \boldsymbol{Exp}$	$::=$	$W \mid (M\ N) \mid (M; N) \mid$ $(\mathrm{ref}_{l,\theta}\ M) \mid (!\ N) \mid (M := N) \mid$ $(\text{if } M \text{ then } N_t \text{ else } N_f) \mid$ $(\text{thread } M) \mid (\text{flow } F \text{ in } M)$

Figure 3.2: Syntax of Expressions

Threads		$::=$	$M \quad (\in \boldsymbol{Exp})$
Pool of Threads	$P \in 2^{\boldsymbol{Exp}}$		
Store	S	$:$	$(\boldsymbol{Ref} \times 2^{\boldsymbol{Pri}} \times \boldsymbol{Typ}) \rightarrow \boldsymbol{Val}$
Configurations		$::=$	$\langle P, S \rangle$

Figure 3.3: Syntax of Configurations

3.2.1 Syntax

The syntax of the security annotations, types, expressions and configurations (see Figures 3.1, 3.2 and 3.3) is mostly the same as that defined on Subsection 2.2.1, with the exception of the usage of flow policies and the flow declaration as a new expression of the language.

Flow Policies

In the previous chapter, a flow policy was used to define a pre-lattice of security levels. It was called the *global flow policy*, since it was unique and regarded all computations. Here we add a way to customize the global flow policy, by locally extending it in order to obtain the notion of *current flow policy* that regards delimited portions of a computation.

Just as in the previous chapter, a flow policy F is a set of pairs of principals, where a pair $(p, q) \in F$, most often written $p \prec q$, 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*".

The Flow Declaration Construct

The flow declaration construct is written (flow F in M), where F is a flow policy, and M is any expression of the language. As we said earlier, the meaning is that M is executed in the context of the current flow policy *extended with* F, and after termination the current flow policy is restored, that is, the scope of F is M.

3.2.2 Semantics

We now define the semantics of the language, a small step operational semantics on configurations. It is mostly the same as the one defined in Subsection 2.2.2, so we adopt the same notations and conventions that were used for the language of the previous chapter (for convenience we repeat them here). The differences are the inclusion of the flow declaration as an evaluation context, and in the notion of flow policy declared by an evaluation context, which decorates the transitions. We omit the set-brackets for pools that are singletons.

Basic Sets and Functions

The following definitions and conventions are the same as the ones adopted in Subsection 2.2.1, which we repeat for completeness.

Given a configuration $\langle P, S\rangle$, we define $\mathrm{dom}(S)$ as the set of decorated references that are mapped by S. We say that a reference name a is fresh in S if it does not occur, with any subscript, in $\mathrm{dom}(S)$, that is if $b_{l,\theta} \in \mathrm{dom}(S)$ implies that $b \neq a$. We denote by $\mathrm{rn}(P)$ the set of decorated reference names that occur in the expressions of P (this notation is extended in the obvious way to expressions). We let $\mathrm{fv}(M)$ be the set of variables occurring free in M.

We restrict our attention to *well formed configurations* $\langle P, S\rangle$ satisfying the following condition for memories, values stored in memories, and thread names:

- $\mathrm{rn}(P) \subseteq \mathrm{dom}(S)$, and
- $a_{l,\theta} \in \mathrm{dom}(S)$ implies $\mathrm{rn}(S(a_{l,\theta})) \subseteq \mathrm{dom}(S)$, and
- all occurrences of a name in a configuration are decorated in the same way.

We denote by $\{x \mapsto W\}M$ the capture-avoiding substitution of W for the free occurrences of x in M. The operation of adding or updating the image of an object z to z' in a mapping Z is denoted $[z := z']Z$.

Flow Contexts

The evaluation contexts of the core language are extended with the flow declaration context (flow F in E), thus giving rise to the evaluation contexts in Figure 3.4.

$$\begin{array}{rcl} \textit{Evaluation Contexts}\quad \mathrm{E} & ::= & [] \mid (\mathrm{E}\ N) \mid (V\ \mathrm{E}) \mid (\mathrm{E}; N) \mid \\ & & (\mathrm{ref}_{l,\theta}\ \mathrm{E}) \mid (!\ \mathrm{E}) \mid (\mathrm{E} := N) \mid (V := \mathrm{E}) \mid \\ & & (\text{if E then } N_t \text{ else } N_f) \mid (\text{flow } F \text{ in E}) \end{array}$$

Figure 3.4: Evaluation Contexts

$$\begin{array}{rcl} \langle \mathrm{E}[((\lambda x.M)\ V)], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[\{x \mapsto V\}M], S\rangle \\ \langle \mathrm{E}[(\text{if } \mathit{tt} \text{ then } N_t \text{ else } N_f)], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[N_t], S\rangle \\ \langle \mathrm{E}[(\text{if } \mathit{ff} \text{ then } N_t \text{ else } N_f)], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[N_f], S\rangle \\ \langle \mathrm{E}[(V; N)], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[N], S\rangle \\ \langle \mathrm{E}[(\varrho x.W)], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[(\{x \mapsto (\varrho x.W)\}\ W)], S\rangle \\ \langle \mathrm{E}[(\text{flow } F \text{ in } V)], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[V], S\rangle \\ \\ \langle \mathrm{E}[(!\ a_{l,\theta})], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[V], S\rangle,\ \textit{where } S(a_{l,\theta}) = V \\ \langle \mathrm{E}[(a_{l,\theta} := V)], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[()], [a_{l,\theta} := V]S\rangle \\ \langle \mathrm{E}[(\mathrm{ref}_{l,\theta}\ V)], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[a_{l,\theta}], [a_{l,\theta} := V]S\rangle,\ \textit{a fresh in } S \\ \langle \mathrm{E}[(\text{thread } N)], S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{N} & \langle \mathrm{E}[()], S\rangle \end{array}$$

$$\frac{\langle \{M\}, S\rangle \xrightarrow[F]{\emptyset} \langle \{M'\}, S'\rangle}{\langle \{M\}, S\rangle \underset{F}{\longrightarrow} \langle \{M'\}, S'\rangle} \qquad \frac{\langle \{M\}, S\rangle \xrightarrow[F]{N} \langle \{M'\}, S'\rangle}{\langle \{M\}, S\rangle \underset{F}{\longrightarrow} \langle \{M', N\}, S'\rangle}$$

$$\frac{\langle P, S\rangle \underset{F}{\longrightarrow} \langle P', S'\rangle \quad \langle P \cup Q, S\rangle \textit{ is well formed}}{\langle P \cup Q, S\rangle \underset{F}{\longrightarrow} \langle P' \cup Q, S'\rangle}$$

Figure 3.5: Semantics

The analysis of whether the information flows that occur in M are to be allowed depends on the flow policies that are declared in the evaluation context where M is executed. We denote by $\lceil \mathrm{E} \rceil$ the flow policy that is permitted by the evaluation context E. It collects all the flow policies that are declared using flow declaration constructs into one single flow policy, using set union. The evaluation contexts that where inherited from the previous chapter do not affect the flow policy of the context. We then obtain the following definition:

Definition 3.2.1 (Flow Policy Declared by an Evaluation Context)**.** *The* flow policy declared by the evaluation context E *is given by* $\lceil \mathrm{E} \rceil$ *where:*

$$\lceil [] \rceil = \emptyset, \qquad \lceil (\text{flow } F \text{ in } \mathrm{E}) \rceil = F \cup \lceil \mathrm{E} \rceil,$$
$$\lceil \mathrm{E}'[\mathrm{E}] \rceil = \lceil \mathrm{E} \rceil, \text{ if } \mathrm{E}' \text{ does not contain flow declarations}$$

Flow Context of a Transition

The labeled transition rules of our semantics are obtained by decorating the (small step) transitions of the core language with the flow policy declared by the evaluation context where they are performed – see Figure 3.5. Then, from a transition of the form

$$\langle \mathrm{E}[M], S \rangle \rightarrow \langle \mathrm{E}[M'], S' \rangle \tag{3.9}$$

where M does not contain any flow declarations, we obtain the following decorated transition:

$$\langle \mathrm{E}[M], S \rangle \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle \mathrm{E}[M'], S' \rangle \tag{3.10}$$

The rule for a flow declaration that ranges over a value (terminated computation) is:

$$\langle \mathrm{E}[(\text{flow } F \text{ in } V)], S \rangle \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle \mathrm{E}[V], S \rangle \tag{3.11}$$

Observe that the transitions do not depend on the flow label F that decorates them. The evaluation of (flow F in M) simply consists in the evaluation of M, annotated with a flow policy that comprises (in the sense of set inclusion) F. The lifespan of the flow declaration terminates when the expression M that is being evaluated terminates (that is, M becomes a value).

When a new thread is created, the flow policy that is permitted by the evaluation context of the parent thread is not kept:

$$\langle \mathrm{E}[(\text{thread } N)], S \rangle \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle \{\mathrm{E}[()], N\}, S \rangle \tag{3.12}$$

As a consequence, the thread that is spawned executes under a more strict flow policy, which means that it is more constrained, from the confidentiality point of view, than the thread that launched it[1].

[1]It would have been possible to let the thread that is spawned inherit the parents flow policy [Almeida Matos & Boudol, 2005]. The rule could be:

$$\langle \mathrm{E}[(\text{thread } M)], S \rangle \xrightarrow[\lceil \mathrm{E} \rceil]{} \langle \{\mathrm{E}[()], (\text{flow } \lceil \mathrm{E} \rceil \text{ in } M)\}, S \rangle \tag{3.13}$$

Here we leave to the programmer the option of declaring a more permissive flow policy for the thread that is created.

Properties of the Semantics

Just as with the language of the previous chapter, one can prove that the semantics preserves the conditions for well-formedness, and that a configuration with a single expression has at most one transition, up to the choice of new names.

The following result states that, if the evaluation of a thread M differs in the context of two distinct memories while not creating two distinct references, this is because M is performing a dereferencing operation, which yields different results depending on the memory.

Lemma 3.2.2 (Splitting Computations)**.**
If $\langle M, S_1\rangle \xrightarrow[F]{N} \langle M_1', S_1'\rangle$ *and* $\langle M, S_2\rangle \xrightarrow[F']{N'} \langle M_2', S_2'\rangle$ *with* $M_1' \neq M_2'$ *and* $\mathrm{dom}(S_2' - S_2) = \mathrm{dom}(S_1' - S_1)$*, then* $N = () = N'$ *and there exist* E *and* $a_{l,\theta}$ *such that* $F = \lceil \mathrm{E} \rceil = F'$*,* $M = \mathrm{E}[(!\ a_{l,\theta})]$*, and* $M_1' = \mathrm{E}[S_1(a_{l,\theta})]$*,* $M_2' = \mathrm{E}[S_2(a_{l,\theta})]$ *with* $S_1' = S_1$ *and* $S_2' = S_2$*.*

Proof. Note that the only rule where the state is used is that for $\mathrm{E}[(!\ a_{l,\theta})]$. By case analysis on the transition $\langle M, S_1\rangle \xrightarrow[F]{N} \langle M_1', S_1'\rangle$. □

The above result is analogous to Lemma 2.2.1, but adds the fact that the flow policies F, F' of the transitions are the same as the one that is declared by the evaluation context E.

3.3 The Non-disclosure Policy

In this section we formally define non-disclosure, the security property that we study in this chapter. We start by showing how the flow relation behind the security pre-lattices can be parameterized by the current flow policies; we then give a bisimulation definition of non-disclosure that relaxes the definition of non-interference given in Section 2.3, using the small-step semantics defined in Section 3.2; finally, we justify the security property with some examples. Explanations will focus mainly on the differences, with respect to the previous chapter, that are introduced here.

3.3.1 Dynamic Security Pre-lattices

So far we have endowed our language with means for expressing dynamically evolving flow policies for dealing with declassification. Information about the flow policy of the evaluation context is now available in the labels that decorate the transitions. These labels are used to extract the appropriate parameter for building the current flow policy, without requiring any change in the security levels that are associated with references – it is only the flow policy that changes. We are then faced with the issue of maintaining a varying pre-lattice structure over a given set of security levels. To achieve this, we consider, at each point of the computation, the *security pre-lattices* that are derived from the current flow policy in the same way as in Section 2.3. We recall their definitions here.

We define the *preorder on security levels* $\preceq_F$ that is determined by the flow policy F. We use the notion of *F-upward closure* of a security level l (defined

by $l \uparrow_F = \{q \mid \exists p \in l.\ p\ F^*\ q\}$) to derive the more permissive *flow relation*:

$$l_1 \preceq_F l_2 \stackrel{\text{def}}{\Leftrightarrow} \forall q \in l_2\ .\ \exists p \in l_1\ .\ p\ F^*\ q \Leftrightarrow (l_1 \uparrow_F) \supseteq (l_2 \uparrow_F) \tag{3.14}$$

We use the above flow relation to define a range of pre-lattices that are determined by a flow policy:

Definition 3.3.1 (Security Pre-lattice)**.** *Given a set* ***Pri*** *of principals and a flow policy F in* ***Pri*** $\times$ ***Pri****, the pair* $(2^{\boldsymbol{Pri}}, \preceq_F)$ *is a* security pre-lattice*, where* meet *($\curlywedge_F$) and* join *($\curlyvee_F$) are given respectively by the union and intersection of the F-upward closures:*

$$l_1 \curlywedge_F l_2 = l_1 \cup l_2 \qquad l_1 \curlyvee_F l_2 = (l_1 \uparrow_F) \cap (l_2 \uparrow_F)$$

We will use the mechanism of extending the flow relation with a flow policy F in the following way: the information flows that are allowed to occur in an expression M placed in a context E[], under a global flow policy G must satisfy the flow relation $\preceq_{G \cup \lceil \mathrm{E} \rceil}$.

3.3.2 A Bisimulation-Based Definition

We now define our security property in terms of the above defined flow relation $\preceq_F$, where F is the current flow policy. The definition of the non-disclosure policy for networks is based on a notion of bisimulation for pools of threads P with respect to a "low" security level. As usual, the bisimulation expresses the requirement that P_1 and P_2 are to be related if, when running over memories that coincide in their low part, they perform the same low changes. Then, if P is shown to be bisimilar to itself, one can conclude that the high part of the memory has not interfered with the low part, i.e., no security leak has occurred. Using the flow policies that were presented earlier, the notion of being low can be extended, thus weakening the condition on the behavior of the threads.

Low-Equality

The notion of "*low-equality*" is the same as in Section 2.3. However, having a flow policy as a parameter takes its full meaning in this chapter, since the current flow policy is not fixed. We recall the definition of low part of a memory and of low-equality of memories with respect to a flow policy F and security level l:

Definition 3.3.2 (Low Part of a Memory)**.** *The* low part of a memory S *with respect to a flow policy F and a security level l is given by:*

$$S \restriction^{F,l} \stackrel{def}{=} \{(a_{k,\theta}, V) \mid (a_{k,\theta}, V) \in S\ \&\ k \preceq_F l\}$$

The low-equality of memories is thus defined:

Definition 3.3.3 (Low-Equality)**.** *The* low-equality between memories S_1 *and* S_2 *with respect to a flow policy F and a security level l is given by:*

$$S_1 =^{F,l} S_2 \stackrel{def}{\Leftrightarrow} S_1 \restriction^{F,l} = S_2 \restriction^{F,l}$$

As we noted earlier, this relation is transitive, reflexive and symmetric. We shall use without notice the fact that:

Remark 3.3.4.

$$F \subseteq F' \text{ and } S_1 =^{F',l} S_2 \text{ implies } S_1 =^{F,l} S_2$$

The Security Property

In the following we denote by $\twoheadrightarrow$ the reflexive closure of the union of the transitions $\underset{F}{\longrightarrow}$, for all F.

Definition 3.3.5 ((G,l)-bisimulation)**.** *A (G,l)-bisimulation is a symmetric relation $\mathcal{R}$ on sets of threads such that:*

$$P_1 \ \mathcal{R} \ P_2 \text{ and } \langle P_1, S_1\rangle \underset{F}{\longrightarrow} \langle P_1', S_1'\rangle \text{ and } S_1 =^{G\cup F,l} S_2 \text{ and } (*)$$

implies

$$\exists P_2', S_2' \ . \ \langle P_2, S_2\rangle \twoheadrightarrow \langle P_2', S_2'\rangle \text{ and } S_1' =^{G,l} S_2' \text{ and } P_1' \ \mathcal{R} \ P_2'$$

where:
$(*)$ $\mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset$

When P_1 performs a transition within the scope of the local flow policy F, it is allowed to read "low"-references from the input memory (S_1 and S_2) according to the current flow policy $G \cup F$. Recall that these references are labeled with a security level l' such that $l' \preceq_{G\cup F} l$. As to the condition imposed on the output memories S_1' and S_2', it suffices[2].

The main difference between this definition and the one of Definition 2.3.5 is the stronger premiss $S_1 =^{G\cup F,l} S_2$. By starting with pairs of memories that are low-equal "to a greater extent", i.e. that coincide in a larger portion of the memory, the condition on the behavior of the program P_2 becomes weaker. As a consequence, this definition potentially relates more programs.

The absence of a condition on the flow policy of the matching move for P_2 enables all expressions without side-effects to be bisimilar, independently of the flow policy that is declared by their evaluation contexts – for example, the programs $((\,);(\,))$ and (flow F in $((\,);(\,))$).

Remark 3.3.6.

- *For any G and l there exists a (G,l)-bisimulation, like for instance the set $\boldsymbol{Val} \times \boldsymbol{Val}$ of pairs of values.*
- *The union of a family of (G,l)-bisimulations is a (G,l)-bisimulation.*

Consequently, there is a largest (G,l)-bisimulation, which is the union of all (G,l)-bisimulations:

[2]Imposing the stronger condition $S_1' =^{G\cup F,l} S_2'$ on the resulting memories would imply that the program

$$(\text{if } (!\ a_H) \text{ then } (\text{flow } H \prec L \text{ in } (b_H := 0)) \text{ else } (\,))$$

would be considered insecure. The reason for this is that the assignment to the reference b_H would be observable at level L, and this would reflect a flow of information from the high reference a_H to require them to be indistinguishable from the point of view of the policy that is restored after the step – the global flow policy G.

Notation 3.3.7. *The largest (G, l)-bisimulation is denoted $\approx_{G,l}$.*

We now define the Non-disclosure policy in the same manner as we did for Non-interference.

Definition 3.3.8 (Non-disclosure with respect to G)**.** *A pool of threads P satisfies the* Non-disclosure *policy (or is* secure *from the point of view of Non-disclosure) with respect to the global flow policy G if it satisfies $P \approx_{G,l} P$ for all security levels l. We then write $P \in \mathcal{NSD}(G)$.*

Using the Flow Declaration

In the examples we give next we assume given two principals H and L, and a global flow policy G consisting of the pair $L \prec H$. We denote, as usual, references with security levels $\{H\}$ or $\{L\}$ simply by a_H or b_L, leaving out the type and the brackets.

The program

$$(b_L := (\text{flow } H \prec L \text{ in } (!\ a_H))) \tag{3.15}$$

is essentially the same as Example 3.3, and is therefore secure. However, the program

$$(\text{if } (!\ a_H) \text{ then } (\text{flow } H \prec L \text{ in } (b_L := tt)) \text{ else } ()) \tag{3.16}$$

is not secure, since the flow declaration does not encompass the declassification of the reference a_H.

As was observed in Section 2.3 with respect to the bisimulation of the previous chapter, the kind of bisimulation that we use here can be considered to be "strong" (as in [Sabelfeld & Sands, 2005]), since each time a transition is matched, we restart the bisimulation game by comparing the resulting pools of threads in the context of any new low-equal memories, rather than continuing with the resulting configurations. This allows us to restore a more restrictive flow policy after a local flow declaration has been used, as in

$$(\text{flow } H \prec L \text{ in } (b_L :=!\ a_H)); (b'_{L'} :=!\ a'_{H'}) \tag{3.17}$$

which is insecure in a context where the flow policy does not consider H' to be lower than L', even if $H = H'$ and $L = L'$.

3.3.3 Properties of Secure Programs

We could state a compositionality result (with respect to set union), as in Proposition 2.3.9. Another property of our notion of security is that if an expression M is secure under the global flow policy $G \cup F$, then the expression (flow F in M) is secure with respect to the global flow policy G:

Proposition 3.3.9.

$$M \in \mathcal{NSD}(G \cup F) \textit{ implies } (\text{flow } F \text{ in } M) \in \mathcal{NSD}(G)$$

Proof. §¡ It is easy to see that if $\mathcal{R}$ is a $(G \cup F, l)$-bisimulation, then the relation

$$\begin{array}{ll} & \{((\text{flow } F \text{ in } M), (\text{flow } F \text{ in } N)) \mid M\ \mathcal{R}\ N\} \\ \cup & \{(V, (\text{flow } F \text{ in } N)) \mid V\ \mathcal{R}\ N\ \&\ V \in \boldsymbol{Val}\} \\ \cup & \{((\text{flow } F \text{ in } M), V) \mid M\ \mathcal{R}\ V\ \&\ V \in \boldsymbol{Val}\} \\ \cup & \{(V, V') \mid V\ \mathcal{R}\ V'\ \&\ V, V' \in \boldsymbol{Val}\} \end{array} \tag{3.18}$$

is a (G, l)-bisimulation. □

To see why the reverse implication is not true[3], suppose that M does not contain any flow declaration, and that $F \neq \emptyset$.

Operationally High Threads

As we did in the previous chapter, we can identify a class of threads that have the property of never performing any change in the "low" part of the memory. These are classified as being "high"according to their behavior:

Definition 3.3.10 (Operationally High Threads)**.** *A set $\mathcal{H}$ of threads is said to be a set of* operationally (G, l)-high threads *if the following holds for any $M \in \mathcal{H}$:*

$$\langle M, S\rangle \xrightarrow[F]{N} \langle M', S'\rangle \text{ implies } S =^{G,l} S'$$

$$\text{and both } M', N \in \mathcal{H}$$

This definition does not differ from that of Definition 2.3.10. Indeed, the low part of the memories is considered with respect to the parameter G, while the flow policy of the transitions of the thread is not taken into account. This is consistent with the definition of bisimulation, where the "observation" of the memories that result from a step are taken from the point of view of the global flow policy G.

Remark 3.3.11.

- *For any G and l there exists a set of operationally (G, l)-high threads, like for instance* ***Val****.*
- *The union of a family of sets of operationally (G, l)-high threads is a set of operationally (G, l)-high threads.*

Therefore, there exists the largest set of operationally (G, l)-high threads:

Notation 3.3.12. *The union of all sets of operationally (G, l)-high threads is denoted by $\mathcal{H}_{G,l}$.*

We say that a thread M is an *operationally (G, l)-high thread* if $M \in \mathcal{H}_{G,l}$. Notice that if $G \subseteq F$, then any operationally (F, l)-high thread is also operationally (G, l)-high. Furthermore, an operationally $\top$-high thread never modifies the memory.

[3]Interestingly, the reverse implication would be true if the definition of (G, l)-bisimulation had been relaxed by comparing the memories that result from the matching steps with respect to the empty flow policy, instead of the global flow policy. However, such a change would mean that non-disclosure would no longer generalize standard non-interference (we will soon see that it does now). It is not clear, however, whether the definition of non-interference itself could be relaxed in the same manner.

Comparison with Non-interference

The Non-disclosure policy is equivalent (up to notational issues) to the Non-interference policy, if we only consider threads that do not contain flow declarations. To see this, let us rewrite the condition for $\mathcal{R}$ to be a bisimulation in the sense of Definition 2.3.5, but using the language of this chapter (excluding the flow declarations):

$$\begin{array}{l} P_1 \ \mathcal{R} \ P_2 \textit{ and } \langle P_1, S_1\rangle \underset{\emptyset}{\rightarrow} \langle P_1', S_1'\rangle \textit{ and } S_1 =^{G\cup\emptyset,l} S_2 \\ \textit{and } (*) \textit{ implies:} \\ \exists P_2', S_2' : \langle P_2, S_2\rangle \twoheadrightarrow \langle P_2', S_2'\rangle \textit{ and } S_1' =^{G,l} S_2' \textit{ and } P_1' \ \mathcal{R} \ P_2' \\ \\ \textit{where:} \\ (*) \ \mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset \end{array} \tag{3.19}$$

For the purpose of this comparison, we shall say that if a pool of threads P satisfies Non-disclosure in the above sense, then $P \in \mathcal{NI}(G, \emptyset)$.

We call *derivative of an expression* M, any expression M' that is attainable from M by a (possibly empty) sequence of small-step transitions.

Definition 3.3.13 (Derivative of an Expression)**.** *We say that an expression M' is a* derivative *of an expression M if and only if*

- *$M' = M$, or*
- *there exist two states S_1 and S_1' and a derivative M'' of M such that, for some F, N:*

$$\langle M'', S_1\rangle \xrightarrow[F]{N} \langle M', S_1'\rangle$$

Proposition 3.3.14. *Consider a pool of threads P whose expressions do not contain flow declarations. Then, $P \in \mathcal{ND}(G)$ if and only if $P \in \mathcal{NI}(G, \emptyset)$.*

Proof. Suppose $P \in \mathcal{ND}(G)$. Then, for all security levels l, there exists a relation $\mathcal{S}$ that is a (G, l)-bisimulation according to Definition 3.3.5, and such that $P \ \mathcal{S} \ P$. Then, we have that

$$\mathcal{S}' \stackrel{\text{def}}{=} \{(Q_1, Q_2) \mid Q_1 \ \mathcal{S} \ Q_2 \ \&\ Q_1, Q_2 \textit{ are derivatives of } P\} \tag{3.20}$$

is also a (G, l)-bisimulation according to Definition 3.3.5 and $P \ \mathcal{S}' \ P$. Since P does not contain flow declarations, then every derivative of P does not contain flow declarations either. Now, suppose that $P_1 \ \mathcal{S}' \ P_2$. Then, if

$$\langle P_1, S_1\rangle \xrightarrow[\emptyset]{N} \langle P_1', S_1'\rangle \tag{3.21}$$

and $S_1 =^{G\cup F,l} S_2$ and $\mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset$. Since $\mathcal{S}'$ is a (G, l)-bisimulation according to Definition 3.3.5,

$$\exists P_2', S_2' \ : \ \langle P_2, S_2\rangle \twoheadrightarrow \langle P_2', S_2'\rangle \tag{3.22}$$

such that $\langle T_1', S_1'\rangle =^{G,l} \langle T_2', S_2'\rangle$ and $P_1' \ \mathcal{S}' \ P_2'$. Hence, $\mathcal{S}'$ is a (G, l)-bisimulation according to 3.19, where $P \ \mathcal{S}' \ P$, and we conclude that $P \in \mathcal{ND}(G)$.

Now suppose $P \in \mathcal{NI}(G, \emptyset)$. Then, for all security levels l, there exists a relation $\mathcal{S}$ that is a (G, l)-bisimulation according to 3.19, and such that $P \; \mathcal{S} \; P$. Now, suppose that $P_1 \; \mathcal{S} \; P_2$. Then, if

$$\langle P_1, S_1 \rangle \xrightarrow[F]{N} \langle P_1', S_1' \rangle \tag{3.23}$$

and $\langle T_1, S_1 \rangle =^{G \cup F, l} \langle T_2, S_2 \rangle$ and $\mathrm{dom}({S_1}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset$, clearly we have $F = \emptyset$. Therefore, since $\mathcal{S}$ is a (G, l)-bisimulation according to 3.19,

$$\exists P_2', S_2' \; : \; \langle P_2, S_2 \rangle \twoheadrightarrow \langle P_2', S_2' \rangle \tag{3.24}$$

where $S_1' =^{G,l} S_2'$ and $P_1' \; \mathcal{S}' \; P_2'$. Therefore, $\mathcal{S}'$ is a (G, l)-bisimulation according to Definition 3.3.5, where $P \; \mathcal{S}' \; P$, and we conclude that $P \in \mathcal{ND}(G)$.

□

It is then clear that all the examples of insecure programs given in the previous chapter do not satisfy non-disclosure. However, if we write these programs in the context of the flow declaration (flow $H \prec L$ in []) (since they all involve leaks from level H to L), they become secure.

3.4 Typing Non-disclosure

In this section we present a type and effect system that only accepts programs that satisfy Non-disclosure. It extends the one that is presented in Section 2.4, so we will restrict the explanations to the features that are introduced here. We start by defining the notation used to express the typing judgments and by explaining their meaning; we then comment on how the flow policies are used to relax the typing conditions, in such a way that declassifying programs can now be accepted; finally, we conclude by giving some properties of the type system, including a Subject Reduction result, and the Soundness theorem.

3.4.1 A Type and Effect System with Flow Policies

The type and effect system that we present here selects secure threads by ensuring the compliance of all information flows to the flow relation that is parameterized with the current flow policy. This consists of the global flow policy G extended with the local flow policy that is valid for each expression, i.e. the one that is declared by the evaluation context where the expression is placed.

The Typing Judgments

As defined in Figure 3.6, the judgments of the type and effect system have the form:

$$\Gamma \vdash_{G,F} M : s, \tau \tag{3.25}$$

The meaning of Γ (the typing environment), M (the expression being typed), s (the security effect of M, including the reading effect, writing effect and termination effect), and G (the global flow policy§¡) is the same as in the previous chapter (see Section 2.4). As for the remaining parameters:

- The flow policy F is the one that is valid in the evaluation context in which M is to be typed, and contributes to the meaning of operations and relations on security levels. It is called the *flow policy of the context*. It is assumed to contain the global flow policy, which is extended with the local flow policies declared by the evaluation context.
- The type τ is the type of the expression. The types we use in this chapter are similar to those of Chapter 2. The syntax (that can be seen in Figure 3.1) is repeated here:

$$\tau, \sigma, \theta \ \in \ \textbf{\textit{Typ}} \ ::= \ t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta \ \mathrm{ref}_l \mid \tau \xrightarrow[F]{s} \sigma$$

  The only difference is in the function type, that records the "latent flow policy", which is assumed to hold when the function is applied to an argument.

In some of the typing rules we use the join operation on security effects:

Definition 3.4.1.

$$s \curlyvee_G s' \quad \stackrel{def}{\Leftrightarrow} \quad (s.r \curlyvee_G s'.r, s.w \curlywedge_G s'.w, s.t \curlyvee_G s'.t)$$

The type and effect system is given in Figure 3.7. Notice that it is syntax directed. We use some abbreviations: instead of the meet $\curlywedge_G$ and join $\curlyvee_G$ with respect to the global flow policy we write $\curlywedge$, $\curlyvee$, respectively; we also omit the global flow policy that appears as subscript of $\vdash_{G,F}$ and simply write $\vdash_F$; whenever we have $\Gamma \vdash M : \langle \bot, \top, \bot \rangle, \tau$, we only write $\Gamma \vdash M : \tau$.

3.4.2 Relaxed Typing Conditions

Let us look at the features of the type system that enable the acceptance of programs that are insecure from the point of view on Non-interference, but secure with respect to the Non-disclosure policy. We therefore focus on the role of the flow policy that appears as a parameter of the judgments.

To type a flow declaration (flow F in M), the expression M is only required to be typable in the context of the current flow policy *extended with* F. To see why typability with respect to a "larger" flow policy is "easier", recall that the conditions that are imposed by the typing rules in fact constrain the flow of information that is encoded by the expressions to comply with the current flow relation. Since flow relations that are parameterized with larger flow policies are weaker, more flows satisfy the conditions that they impose.

The simplest example of a *declassification operation* is the assignment that implements the direct flow in Example 2.15. To obtain a valid program that performs such an operation, it suffices to write:

$$(\text{flow } H \prec L \text{ in } (a_L := (!\ b_H))) \tag{3.26}$$

In fact, to type the above program using Flow, Ass and Der, it is enough that the condition $\{H\} \preceq_{H \prec L} \{L\}$ holds, which clearly is the case. The same principle applies to all examples given in the previous chapter.

As pointed out earlier, it is inevitable that some secure programs are rejected by the type system. For instance, a conditional that tests a high reference

$$(\text{if } (!\ a_H) \text{ then } M \text{ else } N) \tag{3.27}$$

$$\begin{array}{lll} \textit{Typing Environments} & \Gamma & : \quad \boldsymbol{Var} \to \boldsymbol{Typ} \\ \textit{Typing Judgments} & & ::= \quad \Gamma \vdash_{G,F} M : s, \tau \end{array}$$

Figure 3.6: Syntax of Typing Judgments (see also Figure 3.1)

$$[\textsc{Nil}]\ \Gamma \vdash () : \mathsf{unit} \qquad [\textsc{Flow}]\ \frac{\Gamma \vdash_{F \cup F'} M : s, \tau}{\Gamma \vdash_F (\text{flow } F' \text{ in } M) : s, \tau}$$

$$[\textsc{Abs}]\ \frac{\Gamma, x : \tau \vdash_F M : s, \sigma}{\Gamma \vdash (\lambda x.M) : \tau \xrightarrow[F]{s} \sigma} \qquad [\textsc{Rec}]\ \frac{\Gamma, x : \tau \vdash_F s, W : \tau}{\Gamma \vdash (\varrho x.W) : \tau}$$

$$[\textsc{BoolT}]\ \Gamma \vdash tt : \mathsf{bool} \qquad [\textsc{BoolF}]\ \Gamma \vdash ff : \mathsf{bool}$$

$$[\textsc{Var}]\ \Gamma, x : \tau \vdash x : \tau \qquad [\textsc{Loc}]\ \Gamma \vdash a_{l,\theta} : \theta\ \mathrm{ref}_l$$

$$[\textsc{Ref}]\ \frac{\Gamma \vdash_F M : s, \theta \quad s.r \preceq_F l}{\Gamma \vdash_F (\mathrm{ref}_{l,\theta}\ M) : s \curlyvee \langle \bot, l, \bot \rangle, \theta\ \mathrm{ref}_l}$$

$$[\textsc{Der}]\ \frac{\Gamma \vdash_F M : s, \theta\ \mathrm{ref}_l}{\Gamma \vdash_F (!\ M) : s \curlyvee \langle l, \top, \bot \rangle, \theta}$$

$$[\textsc{Ass}]\ \frac{\Gamma \vdash_F M : s, \theta\ \mathrm{ref}_l \quad \Gamma \vdash_F N : s', \theta \qquad \begin{array}{c} s.t \preceq_F s'.w \\ s.r, s'.r \preceq_F l \end{array}}{\Gamma \vdash_F (M := N) : s \curlyvee s' \curlyvee \langle \bot, l, \bot \rangle, \mathsf{unit}}$$

$$[\textsc{Cond}]\ \frac{\Gamma \vdash_F M : s, \mathsf{bool} \quad \begin{array}{c} \Gamma \vdash_F N_t : s_t, \tau \\ \Gamma \vdash_F N_f : s_f, \tau \end{array} \quad s.r \preceq_F s_t.w, s_f.w}{\Gamma \vdash_F (\text{if } M \text{ then } N_t \text{ else } N_f) : s \curlyvee s_t \curlyvee s_f \curlyvee \langle \bot, \top, s.r \rangle, \tau}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash_F M : s, \tau \xrightarrow[F]{s'} \sigma \quad \Gamma \vdash_F N : s'', \tau \qquad \begin{array}{c} s.t \preceq_F s''.w \\ s.r, s''.r \preceq_F s'.w \end{array}}{\Gamma \vdash_F (M\ N) : s \curlyvee s' \curlyvee s'' \curlyvee \langle \bot, \top, s.r \curlyvee s''.r \rangle, \sigma}$$

$$[\textsc{Seq}]\ \frac{\Gamma \vdash_F M : s, \tau \quad \Gamma \vdash_F N : s', \sigma \quad s.t \preceq_F s'.w}{\Gamma \vdash_F (M; N) : s \curlyvee s', \sigma}$$

$$[\textsc{Thr}]\ \frac{\Gamma \vdash_\emptyset M : s, \mathsf{unit}}{\Gamma \vdash_F (\text{thread } M) : \langle \bot, s.w, \bot \rangle, \mathsf{unit}}$$

Figure 3.7: Type and Effect System

can never be accepted as long as it contains low assignments or reference creations in either of the branches M or N. While this restriction rejects all possible control leaks that could be encoded by the conditional, it rejects many other programs that inoffensively write to low references in their branches (see [Volpano & Smith, 1999; Agat, 2000; Sabelfeld & Sands, 2005] for some ways of ensuring, in a simple language, that the branches do not cause any problems). One practical application of the flow declaration construct is to use, instead, the program

$$(\text{flow } H \prec L \text{ in } (\text{if } (!\ a_H) \text{ then } M \text{ else } N)) \tag{3.28}$$

thus neutralizing the effect of the typing restrictions that are imposed by COND.

As a last example, we show that the level to which a program will downgrade the contents of a reference cannot be predicted statically. Let M be the following expression:

$$(\text{if } N \text{ then } (\text{flow } p \prec q \text{ in } (a_{q,\theta} := !\ c_{p,\theta})) \text{ else } (\text{flow } p \prec r \text{ in } (b_{r,\theta} := !\ c_{p,\theta}))) \tag{3.29}$$

Then one can see that M is typable if the condition $s.r \preceq_G \{q, r\}$, where $s.r$ is the reading effect of the boolean N, holds. Since there is no constraint regarding the confidentiality level p of the reference c, the level q or r to which the contents of p are declassified depends only on the value into which the boolean N computes.

3.4.3 Properties of Typed Expressions

Similarly to what we did in Section 2.4.3, we omit the details of the proofs here and refer the reader to Subsection 4.4.3 for the complete proofs.

Subject Reduction

The first main result of this section is Subject Reduction. The Subject Reduction property states that the type of an expression is preserved by reduction. The result does not differ from the one stated in the previous chapter (Theorem 2.4.2). In particular, the conditions that state the "weakening" of the security effects during reduction are the same. This is due to the fact that the flow declarations do not interfere with the construction of the security effects – in the type system of Figure 3.7 they are built with respect to the global flow policy, just like in the type system of Figure 2.7[4].

Theorem 3.4.2 (Subject Reduction)**.**
If for some s, τ we have $\Gamma \vdash_F M : s, \tau$ and $\langle M, S\rangle \xrightarrow[F']{N} \langle M', S'\rangle$ where all $a_{l,\theta} \in \mathrm{dom}(S)$ satisfy $\Gamma \vdash S(a_{l,\theta}) : \theta$, then $\exists s'$ such that $\Gamma \vdash_F M' : s', \tau$ and $s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$. Furthermore, $\exists s''$ such that $\Gamma \vdash_\emptyset N : s'', \mathsf{unit}$ and $s.w \preceq s''.w$.

Proof. The main proof is a case analysis on the transition $\langle M, S\rangle \xrightarrow[F']{N} \langle M', S'\rangle$. See the detailed proof of Theorem 4.4.7 and preceding lemmas. □

[4]In this point, the type system that we present here differs from the one presented in [Almeida Matos & Boudol, 2005]. We comment on this in the discussion on related work (Section 3.5).

Syntactically High Expressions

The notion of a syntactically high expression is defined here as in the previous chapter (Definition 2.4.3). The difference lies only in the fact that it is defined with respect to the current flow policy, while in the previous chapter it is of course defined with respect to the global flow policy.

Definition 3.4.3 (Syntactically High Expressions)**.** *An expression M is* syntactically (F, l)-high *if there exists Γ, s, τ such that $\Gamma \vdash_F M : s, \tau$ with $s.w \not\preceq_F l$. The expression M is a* syntactically (F, l)-high function *if there exists Γ, s, τ such that $\Gamma \vdash M : \tau \xrightarrow[F]{s} \sigma$ with $s.w \not\preceq_F l$.*

We can now state that syntactically high expressions have an operationally high behavior.

Lemma 3.4.4 (High Expressions)**.** *If M is a syntactically (F, l)-high expression, then M is an operationally (F, l)-high thread.*

Proof. See proof of Lemma 4.4.9. □

3.4.4 Soundness

The final result of this chapter, Soundness, states that the type system only accepts expressions that are secure in the sense of Definition 3.3.8, for a global flow policy G. In the remaining of this section we sketch the main definitions and results that can be used to reconstruct a direct proof of this result. A similar proof is given in detail for the richer language of Chapter 4.

We first build a symmetric binary relation between expressions that are typable (in the context of a flow relation F) and whose terminating behaviors do not depend on high references, more precisely, between those that are typable with a low termination effect. It should be such that if the evaluation of two related expressions, in the context of two low-equal memories should split (see Lemma 3.2.2), then the resulting expressions are still in the relation. This relation, namely $\mathcal{T}_{F,low}$, is inductively defined for a flow policy F and a security level *low* in Figure 3.8.

The next proposition states that $\mathcal{T}_{F,low}$ is a kind of "strong bisimulation" with respect to the transition relation $\xrightarrow[F']{N}$.

Proposition 3.4.6 (Strong Bisimulation for Low-Terminating Threads)**.** *If we have $M_1 \; \mathcal{T}_{F,low} \; M_2$ and $\langle M_1, S_1 \rangle \xrightarrow[F']{N} \langle M_1', S_1' \rangle$, with $S_1 =^{F \cup F', low} S_2$ such that a is fresh for S_2 if $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$, then there exist M_2' and S_2' such that $\langle M_2, S_2 \rangle \xrightarrow[F']{N} \langle M_2', S_2' \rangle$ with $M_1' \; \mathcal{T}_{F,low} \; M_2'$, $S_1' =^{F,low} S_2'$ and $\mathrm{dom}(S_1' - S_1) = \mathrm{dom}(S_2' - S_2)$.*

Proof. By induction on the definition of $\mathcal{T}_{F,low}$. □

We now define a larger symmetric binary relation on expressions that are typable in the context of a flow relation F. Similarly to the previous one, it should be possible to relate the results of the computations of two related expressions in the context of two low-equal memories. Given a flow policy F and a security level *low*, we define the binary relation $\mathcal{R}_{F,low}$ on expressions

Definition 3.4.5 ($\mathcal{T}_{F,low}$)**.** *We have that* $M_1\ \mathcal{T}_{F,low}\ M_2$ *if* $\Gamma \vdash_F M_1 : s_1, \tau$ *and* $\Gamma \vdash_F M_2 : s_2, \tau$ *for some* Γ, s_1, s_2 *and* τ *with* $s_1.t \preceq_F low$ *and* $s_2.t \preceq_F low$ *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* $M_2 = (\bar{M}_2; \bar{N})$ *where* $\bar{M}_1\ \mathcal{T}_{F,low}\ \bar{M}_2$*, or*

Clause 4. $M_1 = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ *and* $M_2 = (\mathrm{ref}_{l,\theta}\ \bar{M}_2)$ *where* $\bar{M}_1\ \mathcal{T}_{F,low}\ \bar{M}_2$*, and* $l \not\preceq_F low$*, or*

Clause 5. $M_1 = (!\ \bar{M}_1)$ *and* $M_2 = (!\ \bar{M}_2)$ *where* $\bar{M}_1\ \mathcal{T}_{F,low}\ \bar{M}_2$*, or*

Clause 6. $M_1 = (\bar{M}_1 := \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 := \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{T}_{F,low}\ \bar{M}_2$*, and* $\bar{N}_1\ \mathcal{T}_{F,low}\ \bar{N}_2$*, and* $\bar{M}_1, \bar{M}_2$ *both have type* $\theta\ \mathrm{ref}_l$ *for some* θ *and* l *such that* $l \not\preceq_F low$*, or*

Clause 7. $M_1 = (\text{flow } F' \text{ in } \bar{M}_1)$ *and* $M_2 = (\text{flow } F \text{ in } \bar{M}_2)$ *with* $\bar{M}_1\ \mathcal{T}_{F\cup F',low}\ \bar{M}_2$*.*

Figure 3.8: The relation $\mathcal{T}_{F,low}$

inductively in Figure 3.9. The relation $\mathcal{R}_{F,low}$ is a kind of "strong bisimulation", with respect to the transition relation $\xrightarrow[F']{N}$:

Proposition 3.4.8 (Strong Bisimulation for Low Typable Threads)**.** *Suppose that* $M_1\ \mathcal{R}_{F,low}\ M_2$ *and that* $M_1 \notin \mathcal{H}_{F,low}$*. If* $\langle M_1, S_1\rangle \xrightarrow[F']{N} \langle M_1', S_1'\rangle$*, with* $S_1 =^{F\cup F',low} S_2$ *such that* a *is fresh for* S_2 *if* $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$*, then there exist* M_2' *and* S_2' *such that* $\langle M_2, S_2\rangle \xrightarrow[F']{N} \langle M_2', S_2'\rangle$ *with* $M_1'\ \mathcal{R}_{F,low}\ M_2'$*,* $S_1' =^{F,low} S_2'$*, and* $\mathrm{dom}(S_1' - S_1) = \mathrm{dom}(S_2' - S_2)$*.*

Proof. By induction on the definition of $\mathcal{R}_{G,low}$. □

To conclude the proof of the Soundness Theorem, we must exhibit a bisimulation on pools of threads. Consider the following relation:

Definition 3.4.9 ($\mathcal{R}^\star_{G,low}$)**.** *The relation* $\mathcal{R}^\star_{low}$ *is inductively defined as follows:*

$$a)\ \frac{M \in \mathcal{H}_{G,low}}{\{M\}\ \mathcal{R}^\star_{G,low}\ \emptyset} \qquad b)\ \frac{M \in \mathcal{H}_{G,low}}{\emptyset\ \mathcal{R}^\star_{G,low}\ \{M\}} \qquad c)\ \frac{M_1\ \mathcal{R}_{G,low}\ M_2}{\{M_1\}\ \mathcal{R}^\star_{G,low}\ \{M_2\}}$$

$$d)\ \frac{P_1\ \mathcal{R}^\star_{G,low}\ P_2 \quad Q_1\ \mathcal{R}^\star_{G,low}\ Q_2}{(P_1 \cup Q_1)\ \mathcal{R}^\star_{G,low}\ (P_2 \cup Q_2)}$$

We can now use Strong Bisimulation for Low Typable Threads (Proposition 3.4.8) to prove the following:

Proposition 3.4.10. *The relation* $\mathcal{R}^*_{G,low}$ *is a* (G, low)*-bisimulation.*

Definition 3.4.7 ($\mathcal{R}_{F,low}$)**.** *We have that* $M_1\ \mathcal{R}_{F,low}\ M_2$ *if* $\Gamma \vdash_F M_1 : s_1, \tau$ *and* $\Gamma \vdash_F M_2 : s_2, \tau$ *for some* Γ*,* s_1*,* s_2 *and* τ *and one of the following holds:*

Clause 1'. $M_1, M_2 \in \mathcal{H}_{F,low}$*, or*

Clause 2'. $M_1 = M_2$*, or*

Clause 3'. $M_1 = (\text{if } \bar{M}_1 \text{ then } \bar{N}_t \text{ else } \bar{N}_f)$ *and* $M_2 = (\text{if } \bar{M}_2 \text{ then } \bar{N}_t \text{ else } \bar{N}_f)$ *with* $\bar{M}_1\ \mathcal{R}_{F,low}\ \bar{M}_2$ *and* $\bar{N}_t, \bar{N}_f \in \mathcal{H}_{F,low}$*, or*

Clause 4'. $M_1 = (\bar{M}_1\ \bar{N}_1)$ *and* $M_2 = (\bar{M}_2\ \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{R}_{F,low}\ \bar{M}_2$*, and* $\bar{N}_1, \bar{N}_2 \in \mathcal{H}_{F,low}$*, and* $\bar{M}_1, \bar{M}_2$ *are syntactically* (F, low)*-high functions, or*

Clause 5'. $M_1 = (\bar{M}_1\ \bar{N}_1)$ *and* $M_2 = (\bar{M}_2\ \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{T}_{F,low}\ \bar{M}_2$*, and* $\bar{N}_1\ \mathcal{R}_{F,low}\ \bar{N}_2$*, and* $\bar{M}_1, \bar{M}_2$ *are syntactically* (F, low)*-high functions, or*

Clause 6'. $M_1 = (\bar{M}_1; \bar{N})$ *and* $M_2 = (\bar{M}_2; \bar{N})$ *with* $\bar{M}_1\ \mathcal{R}_{low,F}\ \bar{M}_2$ *and* $\bar{N} \in \mathcal{H}_{F,low}$*, or*

Clause 7'. $M_1 = (\bar{M}_1; \bar{N})$ *and* $M_2 = (\bar{M}_2; \bar{N})$ *with* $\bar{M}_1\ \mathcal{T}_{F,low}\ \bar{M}_2$*, or*

Clause 8'. $M_1 = (\text{ref}_{l,\theta}\ \bar{M}_1)$ *and* $M_2 = (\text{ref}_{l,\theta}\ \bar{M}_2)$ *where* $\bar{M}_1\ \mathcal{R}_{F,low}\ \bar{M}_2$*, and* $l \not\preceq_F low$*, or*

Clause 9'. $M_1 = (!\ \bar{M}_1)$ *and* $M_2 = (!\ \bar{M}_2)$ *where* $\bar{M}_1\ \mathcal{R}_{F,low}\ \bar{M}_2$*, or*

Clause 10'. $M_1 = (\bar{M}_1 := \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 := \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{R}_{F,low}\ \bar{M}_2$*, and* $\bar{N}_1, \bar{N}_2 \in \mathcal{H}_{F,low}$*, and* $\bar{M}_1, \bar{M}_2$ *both have type* $\theta\ \text{ref}_l$ *for some* θ *and* l *such that* $l \not\preceq_F low$*, or*

Clause 11'. $M_1 = (\bar{M}_1 := \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 := \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{T}_{F,low}\ \bar{M}_2$*, and* $\bar{N}_1\ \mathcal{R}_{F,low}\ \bar{N}_2$*, and* $\bar{M}_1, \bar{M}_2$ *both have type* $\theta\ \text{ref}_l$ *for some* θ *and* l *such that* $l \not\preceq_F low$*, 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* $\bar{M}_1\ \mathcal{R}_{F \cup F', low}\ \bar{M}_2$*.*

Figure 3.9: The relation $\mathcal{R}_{F,low}$

To conclude, we now state the main result of the chapter, saying that our type system only accepts threads that can securely run concurrently with other typable threads.

Theorem 3.4.11 (Soundness for Non-disclosure)**.** *Consider a pool of threads P and a global flow policy G. If for any $M \in P$ there exist Γ, s and τ such that $\Gamma \vdash_{G,G} M : s, \tau$, then P satisfies the Non-disclosure policy with respect to the flow policy G.*

Proof. By Clause 2' of Definition 3.4.7, for all choices of security levels *low*, we have that $M \; \mathcal{R}_{G,low} \; M$. By Rule c) of Definition 3.4.9, $\{M\} \; \mathcal{R}^{\star}_{G,low} \; \{M\}$. Since this is true for all $M \in P$, by Rule d) we have that $P \; \mathcal{R}^{\star}_{G,low} \; P$. By Proposition 3.4.10 we conclude that $P \approx_{G,low} P$. □

The above result is compositional, in the sense that it is enough to verify the typability of each thread separately in order to ensure non-disclosure for the whole pool of threads.

3.5 Related Work

A lot of work has been done addressing the concern that once declassification is permitted in a language, it could be inappropriately used to leak more information than what would be considered "safe". This has lead to different forms of *constraints* on the usage of declassification that is permitted to the programmer. Another largely orthogonal motivation is to find suitable tools for expressing declassification and the policies to which it should comply. The challenge here is rather to *enable* the programmer to perform declassifications at will, possibly under the scrutiny of self-imposed specifications.

The recent work [Sabelfeld & Sands, 2005] by Sabelfeld and Sands contains an exhaustive survey on the literature regarding the subject of declassification. In particular, they observe that declassification can be controlled according to four main orthogonal goals as to: *what* information should be released, *when* it should be allowed to happen, *who* should be authorized to use it, and *where* in a system it can be stated. In our comments to related work, we will adopt this classification, under the two main primary goals mentioned above – to constrain declassification, or to enable it – and give a few representative examples.

3.5.1 Constraining Declassification

When? In [Volpano, 2000; Volpano & Smith, 2000], Volpano and Smith restrict downgrading to occur by means of specific "hard" functions. The idea is to impose time-complexity based boundaries on the computations that can reveal some secret value to an attacker. This approach seems more appropriate for applications where the leakage of information should be justified as safe (in cryptography, for instance), but is not flexible enough to serve as a general tool for programmers to leak information at will.

Who? *Robust declassification* is an example of downgrading that is restricted to be performed by authorized subjects. It was proposed and then studied in a series of papers [Myers, 1999; Myers & Liskov, 1997; Zdancewic, 2003; Myers

et al., 2004; Tse & Zdancewic, 2004] by Myers and colleagues. The starting point is a model of security labels that, besides specifying the reading policies of an object, specifies the owners of those policies, and provides operations for manipulating those labels. The idea of robust declassification is to ensure that the policies of an object can only be weakened by a subject that is at least as trustworthy as the owner of that policy. The aim of defining who is entitled to perform declassification operations on objects is orthogonal to ours, though it would be interesting to see whether it could be accommodated in our setting.

What? In another paper [Sabelfeld & Myers, 2004], Sabelfeld and Myers aim at restricting the use of downgrading so that it cannot be exploited by laundering attacks. This involves the use of a downgrading mechanism for values – written declassify (V, L). The use of downgrading operations is only considered safe, according to the notion of *delimited release*, provided that the program does not previously modify data that could influence the value of declassified expressions. For instance, the program

$$((a_H := ff); (b_L := \text{declassify } (a_H, L))) \tag{3.30}$$

is considered insecure. On the other hand, the program

$$((b_L := \text{declassify } (a_H, L)); (c_L := a_H)) \tag{3.31}$$

which is similar to the one of Example 3.17, is considered safe, but is ruled out by the type system. The aim of restricting downgrading of values to release only the "intended" information is different from our purpose of providing a flexible tool for performing declassification.

Where? In [Sabelfeld & Sands, 2005], the question of where in a system information is released is further divided according to two forms of locality: *level locality*, specifying the security levels via which information can be downgraded, and *code locality*, which restricts the places of a program in which declassification can be encoded. A typical example of level locality is *intransitive noninterference* [Rushby, 1992; Roscoe & Goldsmith, 1999; Mantel, 2001], which constrains information to flow according to a non-transitive flow relation. This means that it is possible to restrict the downgrading of information to follow a certain predetermined path of security levels.

3.5.2 Enabling Declassification

As was said in the introduction of this chapter (Section 3.1), the problem that we have addressed is that of *enabling* declassification, rather than that of constraining it as in the above related work. The following examples are in this sense closer to ours.

When? Chong and Myers introduced *declassification policies* [Chong & Myers, 2004] that express the sequence of levels through which a value can be downgraded, provided some conditions are satisfied. These conditions are used in the definition of a generalized noninterference property to mark the steps where declassification occurs. This bears some resemblance to our transitions labeled by a local flow relation, although conditions are rather used to single out sequences of steps that do not involve downgrading operations.

Who? Ferrari *et al.* [Ferrari *et al.*, 1997] proposed to attach "*waivers*" to methods in an object-oriented language to provide a way of making information flow from objects to users. Although the authors claim that "*only privileged methods*" have associated waivers, there seems to be actually no constraint on the flow of information they allow. This idea of a waiver is therefore similar to a local flow relation, though it is not clear whether the notion of "*safe information flow*" that the authors define is similar to our Non-disclosure policy (as far as we can see, this definition does not treat waivers as having a local scope).

What? The work by Li and Zdancewic [Li & Zdancewic, 2005] on *relaxed noninterference* provides some control on what (and how) information is released. To this end, they offer the programmer a way of specifying sophisticated *downgrading policies* by means of an expression in a typed λ-calculus. This control is left in the hands of the programmer, who can freely specify the means by which information is allowed to be downgraded. Their main result is very close in spirit to ours, since "*the security guarantee* [provided by relaxed noninterference] *only assures that the program respects the user's security policies*".

Where? Following [Sabelfeld & Sands, 2005], our flow declaration mechanism can be included in the "where" category, even though it does not attempt to constrain declassification, only to express it. Our answer to the "Where?" question could then be "Anywhere.", both from the level and code locality points of view. Indeed, flow declarations can encompass declassifications in any portions of the program, and set up flows between any security levels[5]. In contrast, in *controlled declassification* Mantel and Sands [Mantel & Sands, 2004] chose to restrict declassification to occur at very precise points of the program, and between levels that are statically fixed.

In spite of the antagonistic choices regarding the flexibility of the declassification tool that is studied, Mantel and Sands work is the closest to ours as regards the way declassification is permitted. We will now make a closer comparison between controlled declassification and non-disclosure. Later on, we conclude this chapter with a discussion on the differences between the type system presented here, and the one presented earlier in [Almeida Matos & Boudol, 2005].

Controlled Declassification vs. Non-disclosure

In addition to a given information flow ordering $\leq$ on security levels, Mantel and Sands consider an "exceptional" non-transitive information flow relation $\rightsquigarrow$ on these levels. The latter relation, which is valid for an entire program, can be seen to extend (in a non-transitive manner) the basic flow ordering in every occurrence of a downgrading assignment:

$$[b_{l'} := a_l] \tag{3.32}$$

[5] A similar construct for introducing flow policies exists in Flow Caml, but with an important difference: there it adds the flow policy F to the global security policy, whereas in our case the declaration is *local*. Such a construct has been mentioned in [Tse & Zdancewic, 2004] under the name of "delegation", but it was not formally studied there. In [Hicks *et al.*, 2005], permission tags are used with the opposite purpose of restricting the range of possible updates to a global flow policy, serving as assumptions for sound execution.

Even if $l \not\leq l'$, this instruction is considered secure provided that the exceptional flow relation includes the pair $l \rightsquigarrow l'$. Using our syntax, it could be expressed as

$$(\text{flow } F \text{ in } (b_{l'} := (!\ a_l))) \tag{3.33}$$

where F would relate l and l' if and only if $l \rightsquigarrow l'$. However, as we will see, the analogy between F and $\rightsquigarrow$ cannot be pushed too far, since both $\rightsquigarrow$ and $(\leq \cup \rightsquigarrow)$ are non-transitive.

In the work of Mantel and Sands, a security property that generalizes noninterference is defined using a kind of bisimulation for programs. For this purpose, the small-step semantics is decorated with two notations: a flag (d or o) that indicates whether the step that is being performed is a downgrading assignment; the "effect" $l' \rightarrow l$, in the case the flag is d, indicating that the level of the variables that are used by the downgrading assignment are l' and l respectively (these can be seen as the reading and writing effect of the operation). For the cases where the program performs a downgrading assignment, the effect is used to relax the condition on the resulting state, while securing that the effect is allowed by the exceptional flow relation, and that no more information than the one expressed by the effect is leaked.

Concerning the declassification construct that is introduced in the language, a first difference is that Mantel and Sands choose to restrict declassification to assignment operations that involve reading and writing single variables. In contrast, the flow declaration allows the programmer to declare the possibility of declassification to occur in any portion of the program, ranging from the fine grained assignment as in Example 3.33, to more complex compositions of commands. Furthermore, while the exceptional flow relation applies to every downgrading command in the program, using our flow declaration one can tailor the basic flow order F to different levels of permissiveness in different portions of the program. In particular, as was shown in Example 3.29, the particular downgrading policy that is actually used in a computation cannot be predicted statically. Using our flow declaration one can have threads run under different flow policies, as in Example 3.6[6].

As regards the security property, Mantel and Sands bisimulation is also based on a decorated small-step semantics that indicates the steps that are subject to a more flexible flow relation. However, since in their work every downgrading assignment is subject to the same exceptional flow relation, a flag is sufficient for this purpose. In contrast, since our flow declaration can express a different "exceptional" flow relation each time it is used, then information on the declared flow policy F must be included as well. Furthermore, in our work, the security property does not make use of the effects of the program – the conditions on the states use F alone. There is thus no exact counterpart of the $l' \rightarrow l$ label that is found in their work, which makes it hard to compare the conditions that are imposed for downgrading steps. It is not clear whether similar information on the effects could be used in our setting, where downgrading operations are not restricted to such precise forms of assignments with very simple effects (a read and a write effect, each coinciding with the security level of a single variable).

Attempting to compare the expressivity of both security properties one can see that they are both expressed by means of an "l-bisimulation", stating that

[6]This feature could be used in a language with code mobility (as the one in the following chapter) to enable threads to migrate with their own flow policies.

a program P is secure if, for all security levels l, P is l-bisimilar to itself. While one can see that controlled declassification is not weaker than non-disclosure, for l-bisimilarity in [Mantel & Sands, 2004] implies l-bisimilarity in our work, the inverse direction is not true, for various reasons:

1. Their l-bisimulation is *strong*, since a step must be matched with exactly one step (as opposed to zero or one steps in our setting). It is well known that security properties resulting from this kind of bisimulations consider timing leaks such as the following as insecure:

$$(\text{if } a_H \text{ then } () \text{ else } ((); ())) \tag{3.34}$$

2. Their l-bisimulation is even stronger when requiring declassification steps to match their effect label. For instance, the program

$$(\text{if } c_\bot \text{ then } [b_H := a_L] \text{ else } b_H := a_L) \tag{3.35}$$

   is considered insecure in their setting (even if $H = L$), but secure (modulo a translation like 3.33) in ours.

3. As to the fact that the exceptional flow relation is non-transitive, the program

$$(\text{flow } A \rightsquigarrow B, B \rightsquigarrow C \text{ in } (b_C := a_A)) \tag{3.36}$$

   is insecure (in general) according to controlled declassification because $A \not\rightsquigarrow C$, while it is considered safe in our setting.

4. Perhaps a more interesting example is the program

$$(\text{flow } H \rightsquigarrow \bot \text{ in } (b_L := a_H)) \tag{3.37}$$

   which is also insecure (in general) according to controlled declassification even though $\bot \ (\leq \cup \rightsquigarrow) \ H$, while it is considered safe in our setting.

A final remark regards the difference in expressivity of the languages and type systems in question: Mantel and Sands consider a simple while language with thread creation, but without reference creation or higher-order expressions, and the type system appears to be more restrictive (it restricts the guards of the loops to $\bot$, for instance), in the line of previous work [Smith & Volpano, 1998; Sabelfeld & Sands, 2000].

Flavors of Non-disclosure

There is a subtlety in the usage of flow declarations that regards the "Where?" question, and that can be explored using type systems. Consider the program of Example 3.15, which is safe according to our notion of Non-disclosure, and which we repeat here:

$$(b_L := (\text{flow } H \prec L \text{ in } (!\ a_H))) \tag{3.38}$$

This program is not typable because the assignment is not performed inside the flow declaration. Another example is

$$((\text{flow } H \prec L \text{ in } (\text{if } (!\ a_H) \text{ then } () \text{ else } \mathsf{loop})); (b_L := 0)) \tag{3.39}$$

where information about H is allowed to be downgraded to the level L. Consequently it can be transmitted via the termination behavior of the conditional and possibly stored in the low level reference b. Notice that the above programs would be accepted if the flow declaration would encompass the whole assignment, in case of Example 3.38, and the whole sequential composition, in case of Example 3.39.

We could have used as in [Almeida Matos & Boudol, 2005] a kind of subsumption in the FLOW rule, on the security effect, to have the above examples accepted.

$$[\text{FLOW'}]\ \frac{\Gamma \vdash_{F\cup F'} M : s, \tau \quad s.r \preceq_{F\cup F'} r \quad s.t \preceq_{F\cup F'} t \quad t \preceq_F r}{\Gamma \vdash_F (\text{flow } F' \text{ in } M) : \langle r, s.w, t\rangle, \tau} \tag{3.40}$$

This rule allows the apparent reading and termination effects of the expression (flow F' in M) to be strengthened, that is, to be considered higher, with respect to F', than the ones of M. Therefore, the security effect of the expression (flow $H \prec L$ in $(!\ a_H)$) that appears in Example 3.38 can be $\langle L, \top, \bot\rangle$, in which case the condition $\{L\} \preceq_G \{L\}$ imposed by the rule ASS is satisfied. However, one should notice that in the typing rule for flow declaration it is not safe to let the writing effect appear to be lower, with respect to F'. Example 3.16 shows why it would be wrong to do so. Therefore, we cannot allow subsumption for the writing level.

The value downgrading facility that is provided by the subsumption rule 3.40 can be used to mimic[7] the declassify (M, l) operator that is used in some languages (see [Myers, 1999; Sabelfeld & Myers, 2004] for instance) for downgrading the value of M to the confidentiality level $l = \{p_1, \ldots, p_n\}$

$$\text{declassify } (M, l) \stackrel{\text{def}}{=} (\text{flow } \{H \prec p_1, \ldots, H \prec p_n\} \text{ in } M) \tag{3.41}$$

where H is a principal that we assume to be greater than every other principal in $\boldsymbol{Pri}$. Indeed, the expression is typable, having l as reading and termination effects:

$$\frac{\Gamma \vdash_G M : s, \tau}{\Gamma \vdash_G \text{declassify } (M, l) : (l, s.w, l), \tau} \tag{3.42}$$

Another example is

$$(b_L := (\text{flow } H \prec L \text{ in encrypt } ((!\ a_H), K))) \tag{3.43}$$

where encrypt is a given encryption function, and K is the encryption key.

Another difference between the type system presented in this chapter and the one presented in [Almeida Matos & Boudol, 2005] is the fact that in the latter one the security effects are built using the extended flow relation, while here they are built with respect to the global flow policy. Using the extended flow relation, the security effects are "weaker" – i.e. more precise. However, this does not necessarily imply greater refinement of the type system, since the

[7] The translation (we give in Example 3.41) is not very precise, since the computation of M occurs under the flow declaration. An accurate translation can be given using the l*et* construct [Boudol, 2005b], but it involves other changes to the typing rules that are beyond the scope of this study. The point is that here we adopt an approach that is "operation-oriented", as opposed to "value-oriented".

restrictions that are imposed over those security effects are taken with respect to the extended flow relation. In fact, one can conjecture that the weaker security effects of [Almeida Matos & Boudol, 2005] could be simplified in the same way as here, without loss of generality.

In the absence of subsumption, the declassifying mechanism that is provided is more in the style of [Sabelfeld & Sands, 2005] that is restricted to assignment operations. By eliminating subsumption from our type system, here we choose to restrict purely to the concept of a flow declaration that enables declassification operations. In this way, we underline the particular style of declassification that is introduced by the flow declarations, as opposed to the approaches that aim at downgrading values.

Chapter 4

Non-disclosure for Mobile Code

This chapter addresses the issue of confidentiality and declassification for global computing in a language-based security perspective. The purpose is to deal with new forms of security leaks, which we call *migration leaks*, introduced by code mobility. We present a generalization of the non-disclosure policy [Almeida Matos & Boudol, 2005] to networks, and a type and effect system for enforcing it. We consider the same language as in the previous chapter, enriched with a notion of domain and a standard migration primitive.

The chapter is organized as follows. In the next section we define a calculus that can express confidentiality problems that arise with network communications. In Section 4.3 we discuss the introduction of multiple flow policies and formulate a non-disclosure property that is suitable for a decentralized setting. In Section 4.4 we develop a type system that only accepts programs satisfying such a property. Finally, we comment on related work.

4.1 Introduction

4.1.1 Information Flow in Code and Resource Mobility

We are interested in controlling information flows in the context of a distributed setting with code mobility. In order to study the security issues arising in such a context, we should consider a language where the notion of locality plays a crucial role: programs are distributed over computation sites, and the possibility of execution or failure of programs cannot be guaranteed by one domain alone – it might, for instance, depend on their location. Now the question is: Could these failures be exploited as covert information flow channels? In the presence of mobile code, the answer is *Yes*. New security leaks, that we call *migration leaks*, arise from the fact that execution or suspension of programs may now depend on secret information.

In order to exemplify the new security leaks that appear in a distributed language with code mobility, we consider for a moment an imperative language with concurrency in the style of [Smith & Volpano, 1998], such that:

- Threads are named (names are ranged over by n), where M^n denotes a thread M named n.
- Threads are placed in *domains* (domain names range over d). The position of each thread in a network is given by a special "location-variable" (for a thread n it is denoted by *pos(n)*).
- Migration is obtained by assigning a new domain name to a variable, where $pos(n) := d$ means that the thread n migrates to domain d (unless the value is already d, which means that it is already there).

One can assume different forms of restrictions on how the location-variables can be accessed. Here we assume that:

- The location-variable *pos(n)* can only be written to by thread n. This means that we only consider *subjective migration*.
- The value of the location-variable *pos(n)* can only be tested for equality with the location of the thread that tests it, i.e., a thread located at d can only test whether $pos(n) = d$. This means that a thread can only know which threads are present in its own domain.

We can then write the following program, where a form of busy waiting (for the arrival of the thread m) is unblocked, depending on the value of a high variable a (assuming that $pos(m) = d$, $pos(n_1) = d_1$, and $pos(n_1) = d_2$):

$$\begin{array}{c}(\text{if } a_H \text{ then } (pos(m) := d_1) \text{ else } (pos(m) := d_2))^m \parallel \\ \parallel (\text{while } pos(m) \neq d_1 \text{ do nil}); (b_L := 1)^{n_1} \parallel \\ \parallel (\text{while } pos(m) \neq d_2 \text{ do nil}); (b_L := 2)^{n_2}\end{array} \tag{4.1}$$

Then, depending on the value of the high variable a, different low assignments would occur to the low variable b.

In the light of the way we treated termination leaks in Chapter 2, such a program would be rejected as soon as a security level would be associated to the location-variable $pos(m)$. This is, very roughly, the approach that we will take in this chapter. However, we consider a more complex language, based on those studied in the previous chapters.

4.1.2 Choosing a Calculus for Global Computing

The languages studied in Chapters 2 and 3 are *local* in the sense that resources are assumed to be accessible to all programs at all times. Such an assumption does not hold in general for networks. In fact, a network can be seen as a collection of computation sites – domains – where resources are only accessible to local programs, and failures can be generated by attempts to access remote resources. To study the problem of whether these forms of failures can give rise to information leaks like the ones exemplified above, we must consider a language where the notion of location of a program and of a resource has an impact on computations.

The design of network models is a whole research area in itself, and there exists a wide spectrum of calculi that focus on different aspects of mobility (find a survey in [Boudol *et al.*, 2002]). Sekiguchi and Yonezawa analyzed in [Sekiguchi & Yonezawa, 1997] various forms of behavior on accesses to remote

references. In particular, resources of "take-away type" move together with the threads that own them, while accesses to remote resources of this type result in failure. In [Boudol, 2004] Boudol adopted this kind of references in the ULM language, where the "mobile references" approach is combined with the principles of reactive synchronous languages [Boussinot & Simone, 1996], and the notion of "suspension" on a remote access replaces that of failure.

Here we are interested in a general and simple framework that addresses the unreliable nature of resource access in networks, as well as trust concerns that are raised when computational entities follow different security orientations. To do this, we add to the language that was considered in the previous chapter a notion of domain and a standard migration primitive that enables the position of resources (references) and programs to change during execution. References are assumed to belong to threads. Migration of a thread to another domain is accompanied by the simultaneous migration of its references to the same domain. Inspired by ULM, we assume that accesses to a reference can only be performed by a program that is located at the same site; remote accesses are suspended until the references become available[1].

To illustrate the suspensive nature of reference access in the language that is studied in this chapter (see Section 4.2), a read access (that is the dereference) to a reference named a, is denoted

$$(?\ a) \tag{4.2}$$

while a write access to a, where a value V is assigned to it, is denoted:

$$(a :=^{?} V) \tag{4.3}$$

If we consider a to belong to thread m, and the access to be performed at the domain d, the behavior of each of the above constructs is similar to

$$(\text{while } pos(m) \neq d \text{ do nil}); (!\ a) \tag{4.4}$$

and to

$$(\text{while } pos(m) \neq d \text{ do nil}); (a := V) \tag{4.5}$$

respectively. This should give an intuition on how migration leaks as the one in Example 4.1 can be encoded (see Section 4.3).

4.1.3 From Non-disclosure to Non-disclosure for Networks

The security properties we have at hand, designed for local computations, are not suitable for treating information flows in a distributed setting with code mobility. In fact, since the location of references in a network can be itself a source of information leaks, the notion of safe program must take this into account. For this purpose, we extend the notion of state, which in the previous chapter coincided with that of a store S, with a mapping T that tracks the position of programs in a network.

Since the visibility of threads is a consequence of the possibility/impossibility of accessing any of its references, we associate to threads a security level that is

[1] For the sake of simplicity, no mechanisms for recovery from suspension-deadlock are considered here. A study of the treatment of information leaks that are inherent to the reactive model can be found in [Almeida Matos *et al.*, 2004].

a lower bound to the security levels of the references that it can own. We then extend the usual indistinguishability relation for memories to states that track the positions of programs in a network. In this way, the formalization of non-disclosure for networks becomes a straightforward generalization of the local bisimulation to one that is defined on a small-step semantics with transitions of the form

$$\langle P, T, S\rangle \underset{F}{\longrightarrow} \langle P', T', S'\rangle \tag{4.6}$$

and where low-equality is defined for states $\langle T, S\rangle$.

4.2 An Imperative Mobile λ-Calculus

We now present the language we will use for studying the security issues introduced by code mobility. It is an extension of the one presented in Chapter 3, an imperative higher-order λ-calculus with thread and reference creation and declassification, enriched with the notion of domain and a basic mobility primitive. We first describe our chosen network model: the configuration of networks and domains, the migration behaviors and the distribution of resources. We then define the syntax and semantics for the calculus at the local and network level. We only comment on the new features of the language, and refer the reader to Sections 2.2 and 3.2 for further explanations. However, we give the full definitions of the syntax of the language, the (small step) semantics for configurations that incorporates a state that now tracks the positions of threads in the network, and also some basic properties of the language. The definitions regarding its syntax are all gathered in page 68, while the ones for the semantics can be found on page 71.

4.2.1 Network Model

A *network* consists of a number of *domains*, places where local computations occur independently. Threads may execute concurrently inside domains, create other threads, and *migrate* to another domain. They can own and create a memory space, a *store* that associates values to *references*, which are addresses of memory containers. These stores move together with the thread they belong to, which means that threads and their local references are, at all times, located in the same domain. However, a thread need not own a reference in order to access it. Read and write operations on references may be performed if and only if the corresponding memory location is present in the domain (otherwise they are implicitly suspended).

4.2.2 Syntax

The language of expressions is an imperative call-by-value λ-calculus that includes dynamic thread and reference creation, a flow declaration construct and a migration primitive. The syntax of the security annotations, types and expressions (see Figures 4.1 and 4.2) is mostly the same as the ones defined in Subsections 2.2.1 and 3.2.1, with the exception of the thread identifiers that appear in the functional type (explained in Section 4.4), the domain, thread and reference names, the suspensive assignment and dereference operations and

the migration primitive which is a new expression of the language. The syntax of networks and configurations is defined in Figure 4.3 and explained below.

Names

We assume given four disjoint countable sets $\boldsymbol{Dom} \neq \emptyset$, $\boldsymbol{Nam}$, $\boldsymbol{Var}$, and $\boldsymbol{Ref}$. *Names* are given to domains ($d \in \boldsymbol{Dom}$), threads ($m, n \in \boldsymbol{Nam}$) and references ($a, b, c$), which we also call *addresses*.

We add annotations (subscripts) to names: *decorated thread names* carry the threads security level, while *decorated reference names* carry the references security level and the type of the values that they can hold, as well as the security level of the thread that owns them. Then, a decorated thread name m_j consists of a pair made of a thread name m and a security level j, while a decorated reference name $m_j.u_{l,\theta}$ is a 5-tuple made of a thread name m, its security level j, a reference identifier u, a type θ and a security level l.

References are lexically associated to the threads that create them: they are of the form $m_j.u$, where u is an identifier given by the thread. Thread and reference names can be created at runtime.

In the following we may omit subscripts whenever they are not relevant, following the convention that the same name has always the same subscript.

Migration and Suspensive Reference Accesses

The language of expressions now includes the migration construct (goto d), where d is a domain name. The meaning is that the thread that executes the migration operation should migrate to the domain d.

The commands (? N) and ($M :=^{?} N$) that appear in the language of expressions replace the dereferencing and assignment operations on references, respectively. The different notation is due to the fact that these operations can potentially suspend, when the reference that is being respectively read or written is not accessible. The notation follows [Boudol, 2004], though here we shall not consider any form of reaction to suspension.

Networks and Configurations

We define *stores* S (which, as in previous chapters, map decorated reference names to values), and *threads*, which are named expressions M^{m_j} (the names are decorated). Threads run concurrently in pools P, which are mappings from decorated thread names to expressions (they can also be seen as sets of threads). Networks are flat juxtapositions of domains, each containing a store and a pool of threads. Thread and domain names are assumed to be distinct; furthermore, references are assumed to be located at the same domain as the thread that owns them (the owner thread's name is a prefix of the reference's name) and to always have the same decorations.

Notice that networks are in fact just a collection of threads and owned references that are running in parallel, and whose executions depend on their relative location. To keep track of the locations of threads and references it suffices to maintain a mapping from thread names to domain names. This is the purpose of T, a *position-tracker*, which is a mapping from a finite set of decorated thread names to domain names. Together with the pool P containing all the

Principals	$p, q \in \boldsymbol{Pri}$		
Security Levels	$l, j, k \subseteq \boldsymbol{Pri}$		
Flow Policies	$F, G \subseteq \boldsymbol{Pri} \times \boldsymbol{Pri}$		
Thread Identifiers	$\check{m}, \check{n} \in \boldsymbol{\check{Nam}}$		
Effects	s	$::=$	$\langle l, l, l \rangle$
Type Variables	t		
Types	$\tau, \sigma, \theta \in \boldsymbol{Typ}$	$::=$	$t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta\ \mathrm{ref}_{l,\check{m}_j} \mid \tau \xrightarrow[G,\check{m}_j]{s} \sigma$

Figure 4.1: Syntax of Security Annotations and Types

Variables	$x, y \in \boldsymbol{Var}$		
Domain Names	$d \in \boldsymbol{Dom}$		
Thread Names	$m, n \in \boldsymbol{Nam}$		
Reference Identifiers	$u, v \in \boldsymbol{Ref}$		
Reference Names	a, b, c	$::=$	$m_j.u$
Decorated Thread Names		$::=$	m_j
Decorated Reference Names		$::=$	$a_{l,\theta}$
Values	$V \in \boldsymbol{Val}$	$::=$	$() \mid x \mid a_{l,\theta} \mid (\lambda x.M) \mid tt \mid ff$
Pseudo-values	$W \in \boldsymbol{Pse}$	$::=$	$V \mid (\varrho x.W)$
Expressions	$M, N \in \boldsymbol{Exp}$	$::=$	$W \mid (M\ N) \mid (M; N) \mid$ $(\mathrm{ref}_{l,\theta}\ M) \mid (?\ N) \mid (M :=^{?} N) \mid$ $(\text{if } M \text{ then } N_t \text{ else } N_f) \mid$ $(\mathrm{thread}_l\ M) \mid (\text{flow } F \text{ in } M) \mid$ $(\text{goto } d)$

Figure 4.2: Syntax of Expressions

Threads		$::=$	$M^{m_j}\ \ (\in \boldsymbol{Exp} \times \boldsymbol{Nam} \times 2^{\boldsymbol{Pri}})$
Pool of Threads	P	$:$	$(\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \rightarrow \boldsymbol{Exp}$
Position-Tracker	T	$:$	$(\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \rightarrow \boldsymbol{Dom}$
Store	S	$:$	$(\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}} \times \boldsymbol{Ref} \times 2^{\boldsymbol{Pri}} \times \boldsymbol{Typ}) \rightarrow \boldsymbol{Val}$
Networks	X, Y	$::=$	$d[\ P, S\ ] \mid X \parallel Y$
Configurations		$::=$	$\langle P, T, S \rangle$

Figure 4.3: Syntax of Configurations

threads in the network, and the store S containing all the references in the network, they form *configurations* $\langle P, T, S\rangle$, on which the evaluation relation is defined in the next subsection. More precisely, given a set $\mathcal{D}$ of domain names in a network, we obtain a configuration of the form $\langle P, T, S\rangle$ from a network $d_1[\ P_1, S_1\ ] \parallel \cdots \parallel d_n[\ P_n, S_n\ ]$, where:

$T = \{m_j \mapsto d_1 | M^{m_j} \in P_1\} \cup \cdots \cup \{m_j \mapsto d_n | M^{m_j} \in P_n\}$,

$P = P_1 \cup \cdots \cup P_n$, and

$S = S_1 \cup \cdots \cup S_n$.

4.2.3 Semantics

We now define the semantics of the language, a small step operational semantics on configurations. It is similar to the one defined in Subsection 3.2.2, so we adopt the same notations and conventions that were used for the language of the previous chapter (we repeat them here). The main differences are the condition on the execution of reference accesses, the naming of references according to the thread that created them, the inclusion of the migration primitive, and the update of the position of threads in the network.

Basic Sets and Functions

The following definitions and conventions extend the ones adopted in Subsections 2.2.1 and 3.2.1.

Given a configuration $\langle P, T, S\rangle$, we call the pair (T, S) the *state* of the configuration. We define $\mathrm{dom}(T)$, $\mathrm{dom}(P)$ and $\mathrm{dom}(S)$ as the sets of decorated names of threads and references that are mapped by T, P and S, respectively. We say that a thread or reference name is fresh in T or S if it does not occur, with any subscript, in $\mathrm{dom}(T)$ or $\mathrm{dom}(S)$, respectively. We denote by $\mathrm{tn}(P)$ and $\mathrm{rn}(P)$ the set of decorated thread and reference names, respectively, that occur in the expressions of P (this notation is extended in the obvious way to expressions). Furthermore, we overload tn and define, for a set R of reference names, the set $\mathrm{tn}(R)$ of thread names that are prefixes of the names in R.

We restrict our attention to *well formed configurations* $\langle P, T, S\rangle$ satisfying the following condition for memories, values stored in memories, and thread names:

- $\mathrm{rn}(P) \subseteq \mathrm{dom}(S)$, and
- $a_{l,\theta} \in \mathrm{dom}(S)$ implies $\mathrm{rn}(S(a_{l,\theta})) \subseteq \mathrm{dom}(S)$, and
- $\mathrm{dom}(P) \subseteq \mathrm{dom}(T)$, and
- $\mathrm{tn}(\mathrm{dom}(S)) \subseteq \mathrm{dom}(T)$, and
- all threads in a configuration have distinct names, and
- all occurrences of a name in a configuration are decorated in the same way.

We denote by $\{x \mapsto W\}M$ the capture avoiding substitution of W for the free occurrences of x in M. The operation of adding or updating the image of an object z to z' in a mapping Z is denoted $[z := z']Z$.

Suspensive Reference Accesses as Evaluation Contexts

The evaluation contexts of the languages of the previous chapters included the evaluation of the arguments to the dereference and the assignment operations. Here these are replaced by the corresponding contexts for the suspensive versions of those two operations. We now have (? E), (E $:=^?$ N) and (V $:=^?$ E) added to the evaluation contexts, thus obtaining the evaluation contexts defined in Figure 4.4. This point is merely notational, since the operational difference between the suspensive and non-suspensive versions of these constructs appears only when the evaluation of the arguments has terminated.

Small Step Semantics

The transitions of our (small step) semantics are defined between configurations. The evaluation rules are defined in Figure 4.5. As usual we omit the set-brackets for pools that are singletons. We start by defining the transitions of a single thread. These are decorated with the thread N^{n_k} that is possibly spawned during that transition, where $N^{n_k} = ()$ if no thread is created. The last three rules use the information contained in the label to add any spawned threads to the pool of threads. By the last rule we can see that the execution of a pool of threads is compositional.

The evaluation of the expressions that depend only on the expression itself remains unchanged with respect to the ones in the previous chapters. As to the evaluation of expressions that might depend on and change the state, we point out that: when a reference is created by a thread m, it is named with a fresh name $m.u$ after the parent thread, for some fresh reference identifier u; the dereference and assignment of a reference that belongs to a thread named n is only performed by a thread named m if m and n are both located at the same domain; when a thread is created, its new fresh name is added to the position-tracker; when the (goto d) statement is executed by a thread m, the position of m in the position-tracker is updated to d.

Summing up, the name of the thread is used in the following rules:

- for the creation of a reference, which is named after the parent thread;
- when a new thread is created, and attached to a domain (namely the parent's one);
- in accesses (read or write) to references, which can only be performed if the accessing thread and the reference are placed in the same domain, as pointers to the position of the corresponding threads.

Properties of the Semantics

Just as with the language of the previous chapter, one can prove that the semantics preserves the conditions for well-formedness, and that a configuration with a single expression has at most one transition, up to the choice of new names.

The following result states that, if the evaluation of a thread M^{m_j} differs in the context of two distinct states while not creating two distinct reference names or thread names, this is because M^{m_j} is performing a dereferencing operation, which yields different results depending on the memory.

$$\begin{array}{lrcl} \textit{Evaluation Contexts} & \mathrm{E} & ::= & [] \mid (\mathrm{E}\ N) \mid (V\ \mathrm{E}) \mid (\mathrm{E}; N) \mid \\ & & & (\mathrm{ref}_{l,\theta}\ \mathrm{E}) \mid (?\ \mathrm{E}) \mid (\mathrm{E} :=^{?} N) \mid (V :=^{?} \mathrm{E}) \mid \\ & & & (\text{if E then } N_t \text{ else } N_f) \mid (\text{flow } F \text{ in E}) \end{array}$$

Figure 4.4: Evaluation Contexts

$$\begin{array}{rcl} \langle \mathrm{E}[((\lambda x.M)\ V)]^{m_j}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[\{x \mapsto V\}M]^{m_j}, T, S\rangle \\ \langle \mathrm{E}[(\text{if } tt \text{ then } N_t \text{ else } N_f)]^{m_j}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[N_t]^{m_j}, T, S\rangle \\ \langle \mathrm{E}[(\text{if } ff \text{ then } N_t \text{ else } N_f)]^{m_j}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[N_f]^{m_j}, T, S\rangle \\ \langle \mathrm{E}[(V; N)]^{m_j}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[N]^{m_j}, T, S\rangle \\ \langle \mathrm{E}[(\varrho x.W)]^{m_j}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[\{x \mapsto (\varrho x.W)\}W]^{m_j}, T, S\rangle \\ \langle \mathrm{E}[(\text{flow } F \text{ in } V)]^{m_j}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[V]^{m_j}, T, S\rangle \end{array}$$

$$\frac{T(n_k) = T(m_j)}{\langle \mathrm{E}[(?\ n_k.u_{l,\theta})]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[V]^{m_j}, T, S\rangle},\ \textit{where } S(n_k.u_{l,\theta}) = V$$

$$\frac{T(n_k) = T(m_j)}{\langle \mathrm{E}[(n_k.u_{l,\theta} :=^{?} V)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[()]^{m_j}, T, [n_k.u_{l,\theta} := V]S\rangle}$$

$$\begin{array}{rcl} \langle \mathrm{E}[(\mathrm{ref}_{l,\theta}\ V)]^{m_j}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[a_{l,\theta}]^{m_j}, T, [a_{l,\theta} := V]S\rangle,\ a = m_j.u \textit{ fresh in } S \\ \langle \mathrm{E}[(\mathrm{thread}_k\ N)]^{m_j}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{N^{n_k}} & \langle \mathrm{E}[()]^{m_j}, [n_k := T(m_j)]T, S\rangle,\ n \textit{ fresh in } T \\ \langle \mathrm{E}[(\text{goto } d)]^{m_j}, T, S\rangle & \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} & \langle \mathrm{E}[()]^{m_j}, [m_j := d]T, S\rangle \end{array}$$

$$\frac{\langle \{M^{m_j}\}, T, S\rangle \xrightarrow[F]{\emptyset} \langle \{M'^{m_j}\}, T', S'\rangle}{\langle \{M^{m_j}\}, T, S\rangle \underset{F}{\longrightarrow} \langle \{M'^{m_j}\}, T', S'\rangle} \quad \frac{\langle \{M^{m_j}\}, T, S\rangle \xrightarrow[F]{N^{n_k}} \langle \{M'^{m_j}\}, T', S'\rangle}{\langle \{M^{m_j}\}, T, S\rangle \underset{F}{\longrightarrow} \langle \{M'^{m_j}, N^{n_k}\}, T', S'\rangle}$$

$$\frac{\langle P, T, S\rangle \underset{F}{\longrightarrow} \langle P', T', S'\rangle \quad \langle P \cup Q, T, S\rangle \textit{ is well formed}}{\langle P \cup Q, T, S\rangle \underset{F}{\longrightarrow} \langle P' \cup Q, T', S'\rangle}$$

Figure 4.5: Semantics

Lemma 4.2.1 (Splitting Computations)**.**
If $\langle M^{m_j}, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle {M_1}'^{m_j}, T_1', S_1'\rangle$ *and* $\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N'^{n_k}} \langle {M_2}'^{m_j}, T_2', S_2'\rangle$ *with* ${M_1}'^{m_j} \neq {M_2}'^{m_j}$ *and* $\mathrm{dom}(T_2' - T_2) = \mathrm{dom}(T_1' - T_1)$, $\mathrm{dom}(S_2' - S_2) = \mathrm{dom}(S_1' - S_1)$, *then* $N^{n_k} = () = N'^{n_k}$ *and there exist* E *and* $a_{l,\theta}$ *such that* $F = \lceil \mathrm{E} \rceil = F'$, $M = \mathrm{E}[(?\ a_{l,\theta})]$, *and* $M' = \mathrm{E}[S_1(a_{l,\theta})]$, $M'' = \mathrm{E}[S_2(a_{l,\theta})]$ *with* $\langle T_1', S_1'\rangle = \langle T_1, S_1\rangle$ *and* $\langle T_2', S_2'\rangle = \langle T_2, S_2\rangle$.

Proof. Note that the only rule where the state is used is that for $\mathrm{E}[(?\ a_{l,\theta})]$. By case analysis on the transition $\langle M^{m_j}, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle {M_1}'^{m_j}, T_1', S_1'\rangle$. □

Similarly to Lemmas 2.2.1 and 3.2.2, the creation of new reference and thread names are ignored by means of the conditions $\mathrm{dom}(T_2' - T_2) = \mathrm{dom}(T_1' - T_1)$ and $\mathrm{dom}(S_2' - S_2) = \mathrm{dom}(S_1' - S_1)$.

4.3 The Non-disclosure Policy for Networks

In this section we formally define non-disclosure for networks, the security property that we study in this chapter. We start by defining the security pre-lattices in terms of a flow relation that is parameterized by the contexts flow policy, and discuss the meaning of a "thread flow policy"; then we exhibit an indistinguishability relation on states (that include position-trackers); we then give a bisimulation definition of non-disclosure for networks, using the small-step semantics defined in Section 4.2; finally, we justify the security property with some examples and give some properties of secure programs.

4.3.1 Global Security Pre-Lattices

As in the previous chapter, we have endowed our language with means for expressing dynamically evolving flow policies for dealing with declassification. We consider, at each point of the computation, the *security pre-lattices* that are derived from a flow policy in a similar way to what was done in Sections 2.3 and 3.3. We repeat the definitions here.

We define the *preorder on security levels* $\preceq_F$ that is determined by the flow policy F. We use the notion of *F-upward closure* of a security level l (defined as before by $l \uparrow_F = \{q \mid \exists p \in l.\ p\ F^*\ q\}$) to derive the more permissive *flow relation*:

$$l_1 \preceq_F l_2 \stackrel{\mathrm{def}}{\Leftrightarrow} \forall q \in l_2\ .\ \exists p \in l_1\ .\ p\ F^*\ q \Leftrightarrow (l_1 \uparrow_F) \supseteq (l_2 \uparrow_F) \tag{4.7}$$

We use the above flow relation to define a range of pre-lattices that are determined by a flow policy:

Definition 4.3.1 (Security Pre-lattice)**.** *Given a set* ***Pri*** *of* principals *and a flow policy F in* $\boldsymbol{Pri} \times \boldsymbol{Pri}$*, the pair* $(2^{\boldsymbol{Pri}}, \preceq_F)$ *is a* security pre-lattice*, where* meet *($\curlywedge_F$) and* join *($\curlyvee_F$) are given respectively by the union and intersection of the F-upward closures with respect to F:*

$$l_1 \curlywedge_F l_2 = l_1 \cup l_2 \qquad l_1 \curlyvee_F l_2 = (l_1 \uparrow_F) \cap (l_2 \uparrow_F)$$

We have seen in the previous chapters that, in the absence of flow declarations, if G is the (global) flow policy to which every thread complies, then $\preceq_G$ determines the allowed flows of information. However, here we are assuming that every thread in a network has its own flow policy, so an issue arises as to which flow policy should be considered for validating different information flows. A practical and conservative approach could be to simply consider the minimum flow relation $\preceq$, which clearly all security pre-lattices satisfy. Nevertheless, for the sake of generality, we can admit the existence of such a global flow policy G, that is some sort of intersection over all the threads' flow policies, and use it to parameterize the flow relation.

In summary, we will use the mechanism of extending the flow relation with a flow policy in the same ways as in Section 3.3: if G is the global flow policy (which clearly can be taken to be $\emptyset$), the information flows that are allowed to occur in an expression M placed in a context E[] must satisfy the flow relation $\preceq_{G \cup \lceil \mathrm{E} \rceil}$.

Imposing a Flow Policy

There is another possible view of what it means for a thread to have a flow policy. The difference between the two views is subtle, so let us illustrate it with an example, where two threads m and n have flow policies $F_m = \{p \prec q\}$ and $F_n = \emptyset$ respectively:

$$(m.u_{\{p\}} :=^{?} n.u_{\{p\}}) \tag{4.8}$$

Adopted view. If F is the flow policy of a thread, then every flow that occurs in the network should comply with F.

In other words, the programmer of each thread has his/her own understanding of which security pre-lattice computations should comply with, independently of one another's. According to this view the above flow can be considered harmless, since from the point of view of both F_m and F_n, information in $n.u$ should be allowed to flow to $m.u$. In fact, we have $\{p\} \uparrow_{F_m} \subseteq \{p\} \uparrow_{F_m}$ and $\{p\} \uparrow_{F_n} \subseteq \{p\} \uparrow_{F_n}$, which respectively implies that $\{p\} \preceq_{F_m} \{p\}$ and that $\{p\} \preceq_{F_n} \{p\}$.

Alternative view. If F is the flow policy of a thread n, then all the flows that occur in the network and *that regard n's references* must comply with F.

Since the security levels to be compared have different origins, i.e., their meaning is given by different flow policies, it is not straightforward to see how one should parameterize a flow relation to check whether the above flow should be valid. However, by considering the upward closures of the security levels that are attached to references, with respect to the flow policy of the thread that owns the references, one can see that this flow should be illegal. In fact, $\{p\} \uparrow_{F_m} \not\subseteq \{p\} \uparrow_{F_n}$. In other words, the information in $n.u$ would be leaking to a reference that is accessible by q.

To implement this view, one could require all security levels to be upward closed w.r.t. their thread's flow policy. For instance, if the programmer of a thread believes that principal p can transmit information to q, that is $p \prec q$, then he/she should not include p as an authorized reader of a reference if he/she is not willing to disclose it to q as well.

Here we do not constrain the security levels that are given to references to be upward closed in any way, and leave to the programmer the option of expressing his/her flow policy via the security levels he/she gives to references that are created by their threads. In practice, adding principals to a security level restricts write accesses and enlarges read accesses. Regardless of the choice made for the security levels, flows that comply with the global flow policy G also comply with every flow policy that extends it, for

$$\forall F \;.\; G \subseteq F \;.\; l_1 \uparrow_G \supseteq l_2 \uparrow_G \;\; \textit{implies}\; l_1 \uparrow_F \supseteq l_2 \uparrow_F \tag{4.9}$$

or in other words:

$$\forall F \;.\; G \subseteq F \;.\; l_1 \preceq_G l_2 \;\textit{implies}\; l_1 \preceq_F l_2 \tag{4.10}$$

This observation allows us to have a general notion of security level that is simply that of a set of principals, independent of the threads flow policies.

4.3.2 A Bisimulation-Based Definition

We now define our security property in terms of the above defined flow relation $\preceq_F$, where F is the current flow policy.

Low-equality

The notion of "*low-equality*" is similar to the one in Section 3.3. However, as we will see towards the end of this section, the position of a thread in the network can reveal information about the values in the memory. For this reason, we must use a notion of low-equality that is extended to states. The intuition is that a thread can access a low reference if and only if it is located at the same domain as the thread that owns it. Threads that own low references can then be seen as "low threads". We are interested in states where low threads are co-located. Low-equality on states is defined pointwise, for a security level considered as "low", as follows:

Definition 4.3.2 (Low Part of a State)**.** *The* low part of a state $\langle T, S\rangle$ *is composed of the* low part of a memory S *and* of the position-tracker T *with respect to a flow policy F and a security level l, which are given by:*

$$T \restriction^{F,l} \;\stackrel{def}{=}\; \{(n_k, d) \mid (n_k, d) \in T \;\&\; k \preceq_F l\}$$

$$S \restriction^{F,l} \;\stackrel{def}{=}\; \{(a_{k,\theta}, V) \mid (a_{k,\theta}, V) \in S \;\&\; k \preceq_F l\}$$

We say that two states are "low"-equal if they coincide in their "low" part:

Definition 4.3.3 (Low-Equality)**.** *The* low-equality between states $\langle T_1, S_1\rangle$ *and* $\langle T_2, S_2\rangle$ *with respect to a flow policy F and a security level l is given by the conjunction of the* low-equality between the memories S_1 *and* S_2 *and the* low-equality between the position-trackers T_1 *and* T_2 *with respect to the same security level and flow policy:*

$$\langle T_1, S_1\rangle =^{F,l} \langle T_2, S_2\rangle \;\stackrel{def}{\Leftrightarrow}\; T_1 \restriction^{F,l} = T_2 \restriction^{F,l} \;\textit{and}\; S_1 \restriction^{F,l} = S_2 \restriction^{F,l}$$

This relation is also transitive, reflexive and symmetric. We shall use without notice the fact that:

Remark 4.3.4.

$$F \subseteq F' \text{ and } \langle T_1, S_1\rangle =^{F',l} \langle T_2, S_2\rangle \text{ implies } \langle T_1, S_1\rangle =^{F,l} \langle T_2, S_2\rangle$$

The Security Property

Now we define a bisimulation for networks, which can be used to relate networks with the same behavior over low parts of the states. In the following we denote by $\twoheadrightarrow$ the reflexive closure of the union of the transitions $\underset{F}{\longrightarrow}$, for all F.

Definition 4.3.5 ((G,l)-bisimulation and $\approx_{G,l}$)**.** *A* (G,l)-bisimulation *is a symmetric relation* $\mathcal{R}$ *on sets of threads such that:*

$P_1 \ \mathcal{R}\ P_2$ & $\langle P_1, T_1, S_1\rangle \underset{F}{\longrightarrow} \langle P_1', T_1', S_1'\rangle$ & $\langle T_1, S_1\rangle =^{G\cup F,l} \langle T_2, S_2\rangle$
and $(*)$ *implies*
$\exists T_2', P_2', S_2' \ . \ \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2', S_2'\rangle$ & $\langle T_1', S_1'\rangle =^{G,l} \langle T_2', S_2'\rangle$ & $P_1' \ \mathcal{R}\ P_2'$

where:
$(*)$ $\mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset$ *and* $\mathrm{dom}(T_1{}' - T_1) \cap \mathrm{dom}(T_2) = \emptyset$

Remark 4.3.6.

- *For any* G *and* l *there exists a* (G,l)*-bisimulation, like for instance the set* $\boldsymbol{Val} \times \boldsymbol{Val}$ *of pairs of values.*
- *The union of a family of* (G,l)*-bisimulations is a* (G,l)*-bisimulation.*

Consequently, there is a largest (G,l)-bisimulation, which is the union of all (G,l)-bisimulations:

Notation 4.3.7. *The largest* (G,l)*-bisimulation is denoted* $\approx_{G,l}$*.*

Intuitively, our security property must state that, at each computation step performed by some thread in a network, the information flow that occurs respects the global flow policy, extended with the flow policy (F) that is declared by the context where the command is executed.

Definition 4.3.8 (Non-disclosure for Networks with respect to G)**.** *A pool of threads* P *satisfies the* Non-disclosure for Networks *policy (or is* secure *from the point of view of Non-disclosure for Networks) with respect to the global flow policy* G *if it satisfies* $P \approx_{G,l} P$ *for all security levels* l*. We then write* $P \in \mathcal{NDN}(G)$*.*

The non-disclosure definition differs from that of Definition 3.3.8 in that the position of the "low threads" is treated as "low-information".

Examples of Insecure Migrations

We leave out the stores in the following examples. Suspension on an access to an absent reference can be unblocked by other threads. This allows us to write a program that is similar to Example 2.5, where non-termination is encoded by a suspended access and unblocked by migration:

$$\begin{array}{c} d[(\text{if } a_H \text{ then } (\text{goto } d_1) \text{ else } (\text{goto } d_2))^{n_k}] \parallel \\ d_1[((n_k.x_\top :=^? 0); (m_{1_{j_1}}.y_L :=^? 1))^{m_{1_{j_1}}}] \parallel \\ d_2[((n_k.x_\top :=^? 0); (m_{2_{j_2}}.y_L :=^? 2))^{m_{2_{j_2}}}] \end{array} \tag{4.11}$$

Then, depending on the value of the high reference a, different low assignments would occur to the low references $m_1.y_L$ and $m_2.y_L$. The same example can show a potential leak of information about the positions of the threads m_1 and m_2 via their own low references $m_1.y_L, m_2.y_L$.

An analogous but more direct example shows that the mere arrival of a thread and its references to another domain might trigger a suspended low assignment:

$$\begin{array}{c} d[(\text{if } a_H \text{ then } (\text{goto } d_1) \text{ else } (\text{goto } d_2))^{n_k}] \parallel \\ d_1[(n_k.y_L :=^? 1)^{m_{1_{j_1}}}] \parallel \\ d_2[(n_k.y_L :=^? 2)^{m_{2_{j_2}}}] \end{array} \tag{4.12}$$

We have given an intuition on why the position of a thread in a network can be viewed as information that is accessible via the references that the thread owns. The security level of that information is thus a lower bound on the security levels of its references, and is represented by the security level that is attached to each thread. Consequently, there are other forms of security leaks regarding the position of threads in the network that must be rejected as well.

The previous examples show how migration of a thread can result in an information leak from a high reference to a lower one via an "observer" thread. It is the ability of the observer thread to detect the presence of the first thread that allows the leak. However, one must also prevent the thread itself from revealing information about its own position, like via a low assignment that follows a remote assignment

$$d[((n.u_\top :=^? 0); (b_L :=^? 0))^{m_H}] \tag{4.13}$$

or a remote dereference

$$d[((?\ n.u_\top); (b_L :=^? 0))^{m_H}] \tag{4.14}$$

or when the low assignment itself is remote:

$$d[(b_L :=^? 0)^{m_H}] \tag{4.15}$$

4.3.3 Properties of Secure Programs

We could state a compositionality result (with respect to set union), as in Proposition 2.3.9. Another property of our notion of security is that if an expression M is secure under the global flow policy $G \cup F$, then the expression (flow F in M) is secure with respect to the global flow policy G, as in Proposition 3.3.9.

Operationally High Threads

As we did in the previous chapters, we can identify a class of threads that have the property of never performing any change in the "low" part of the memory. These are classified as being "high" according to their behavior[2]:

Definition 4.3.9 (Operationally High Threads)**.** *A set $\mathcal{H}$ of threads is said to be a set of* operationally (F,l)-high threads *if the following holds for any $M^{m_j} \in \mathcal{H}$:*

$$\langle M^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle M'^{m_j}, T', S'\rangle \text{ implies } \langle T, S\rangle =^{F,l} \langle T', S'\rangle$$

$$\text{and both } M'^{m_j}, N^{n_k} \in \mathcal{H}$$

This definition is similar to that of Definition 2.3.10. Indeed, the low part of the states is considered with respect to the parameter F, while the flow policy of the transitions of the thread is not taken into account.

Remark 4.3.10.

- *For any F and l there exists a set of operationally (F,l)-high threads, like for instance $\{V^{m_j} \mid V \in \boldsymbol{Val}\}$.*
- *The union of a family of sets of operationally (F,l)-high threads is a set of operationally (F,l)-high threads.*

Therefore, there exists the largest set of operationally (F,l)-high threads:

Notation 4.3.11. *The union of all sets of operationally (F,l)-high threads is denoted by $\mathcal{H}_{F,l}$.*

We say that a thread M^{m_j} is an *operationally (F,l)-high thread* if $\{M^{m_j}\} \in \mathcal{H}_{F,l}$. Notice that if $F' \subseteq F$, then any operationally (F,l)-high thread is also operationally (F',l)-high.

Comparison with Non-disclosure

The Non-disclosure for Networks policy that is restricted to networks where only one domain exists is equivalent (up to notational issues) to the Non-disclosure policy, if we only consider threads that do not contain migration instructions. To see this, let us rewrite the condition for $\mathcal{R}$ to be a bisimulation in the sense of Definition 3.3.5, but using the language of this chapter (excluding the migration instructions):

$$\begin{array}{l} P_1 \ \mathcal{R}\ P_2 \text{ and } \langle P_1, T_1, S_1\rangle \underset{F}{\longrightarrow} \langle P_1', T_1, S_1'\rangle \text{ and } S_1 =^{G\cup F,l} S_2 \\ \text{and } (*) \text{ and } (**) \text{ implies:} \\ \exists P_2', S_2' : \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2, S_2'\rangle \text{ and } S_1' =^{G,l} S_2' \text{ and } P_1' \ \mathcal{R}\ P_2' \\ \text{where:} \\ (*)\ \mathrm{dom}({S_1}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset \\ (**)\ \mathcal{I}(T_1) = \mathcal{I}(T_2) = \{d\} \end{array} \tag{4.16}$$

[2]The notion of "operationally high thread" that we define here should not not be confused with the notion of "high thread". The former refers to the security level that is associated with a thread, while the latter refers to the changes that the thread performs on the state.

For the purpose of this comparison, we shall say that if a pool of threads P satisfies Non-disclosure in the above sense, then $P \in \mathcal{ND}(G, d)$.

We shall use the notion of *derivative of an expression* M, as in Definition 3.3.13:

Definition 4.3.12 (Derivative of an Expression)**.** *We say that an expression M' is a* derivative *of an expression M if and only if*

- $M' = M$, *or*
- *there exist two states $\langle T_1, S_1\rangle$ and $\langle T_1', S_1'\rangle$ and a derivative M'' of M such that, for some F, N^{n_k}:*

$$\langle M'', T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle M', T_1', S_1'\rangle$$

Proposition 4.3.13. *Consider a pool of threads P whose expressions do not contain migration instructions. Then, if we consider a network with a single domain d, we have that $P \in \mathcal{NDN}(G)$ if and only if $P \in \mathcal{ND}(G, d)$.*

Proof. Suppose $P \in \mathcal{NDN}(G)$. Then, for all security levels l, there exists a relation $\mathcal{S}$ that is a (G, l)-bisimulation according to Definition 4.3.5, and such that $P \; \mathcal{S} \; P$. Then, we have that

$$\mathcal{S}' \stackrel{\text{def}}{=} \{(Q_1, Q_2) \mid Q_1 \; \mathcal{S} \; Q_2 \;\&\; Q_1, Q_2 \textit{ are derivatives of } P\} \tag{4.17}$$

is also a (G, l)-bisimulation according to Definition 4.3.5 and $P \; \mathcal{S}' \; P$. Since P does not contain migration instructions, then every derivative of P does not contain migration instructions either. Now, suppose that $P_1 \; \mathcal{S}' \; P_2$. Then, if

$$\langle P_1, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle P_1', T_1, S_1'\rangle \tag{4.18}$$

and $S_1 =^{G\cup F, l} S_2$ and $\mathrm{dom}({S_1}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset$, clearly we also have $\langle T_1, S_1\rangle =^{G\cup F, l} \langle T_2, S_2\rangle$ and $\mathcal{I}(T_1) = \mathcal{I}(T_2) = \{d\}$. Therefore, since $\mathcal{S}'$ is a (G, l)-bisimulation according to Definition 4.3.5,

$$\exists T_2', P_2', S_2' \; : \; \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2', S_2'\rangle \tag{4.19}$$

such that $\langle T_1', S_1'\rangle =^{G, l} \langle T_2', S_2'\rangle$ and $P_1' \; \mathcal{S}' \; P_2'$. Since P_2 does not contain migration instructions, then $T_2' = T_2$. Clearly, $S_1' =^{G, l} S_2'$. Therefore, $\mathcal{S}'$ is a (G, l)-bisimulation according to 4.16, where $P \; \mathcal{S}' \; P$, and we conclude that $P \in \mathcal{ND}(G, d)$.

Now suppose $P \in \mathcal{ND}(G, d)$. Then, for all security levels l, there exists a relation $\mathcal{S}$ that is a (G, l)-bisimulation according to 4.16, and such that $P \; \mathcal{S} \; P$. Now, suppose that $P_1 \; \mathcal{S} \; P_2$. Then, if

$$\langle P_1, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle P_1', T_1, S_1'\rangle \tag{4.20}$$

and $\langle T_1, S_1\rangle =^{G\cup F, l} \langle T_2, S_2\rangle$ and $\mathrm{dom}({S_1}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset$, clearly we also have $S_1 =^{G\cup F, l} S_2$ and $\mathcal{I}(T_1) = \mathcal{I}(T_2) = \{d\}$. Therefore, since $\mathcal{S}$ is a (G, l)-bisimulation according to 4.16,

$$\exists P_2', S_2' \; : \; \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2, S_2'\rangle \tag{4.21}$$

where $S'_1 =^{G,l} S'_2$ and $P'_1 \; \mathcal{S}' \; P'_2$. Clearly, $\langle T'_1, S'_1 \rangle =^{G,l} \langle T'_2, S'_2 \rangle$. Therefore, $\mathcal{S}'$ is a (G, l)-bisimulation according to Definition 4.3.5, where $P \; \mathcal{S}' \; P$, and we conclude that $P \in \mathcal{NDN}(G)$.

□

It is then clear that all the examples of insecure programs given in the previous chapter, when placed in a single domain, do not satisfy non-disclosure for networks.

4.4 Typing Non-disclosure for Networks

In this section we present a type and effect system that only accepts programs that satisfy Non-disclosure for Networks. It extends the one that is presented in Section 3.4, so we will focus the explanations on the features that are introduced here. We start by defining the notation used to express the typing judgments and by explaining their meaning; we then comment on the typing conditions used in the typing rules, by giving examples of migration leaks that illustrate why each condition is necessary; finally, we conclude by giving some properties of the type system, including Subject Reduction and Soundness theorems.

4.4.1 A Type and Effect System with Thread Identifiers

The type and effect system that we present here selects secure threads by ensuring the compliance of all information flows to the flow relation that rules in each point of the program. As in the previous chapters, it constructively approximates the *effects* of each expression, which include information on the security levels of the references on which termination or non-termination of the computations might depend.

A key observation is that here non-termination of a computation might arise from an attempt to access a *foreign* reference. In order to distinguish the threads that own each expression and reference, we associate unique identifiers $\check{m}, \check{n} \in \boldsymbol{\check{Nam}}$ to names of already existing threads, as well as to the unknown thread name '?' for those that are created at runtime.

It should now be clear that information on which the position of a thread n might depend can leak when another thread simply attempts to access one of n's references. For this reason, we interpret the threads' security level – in fact it represents its "visibility" level – as a lower bound to the references that it can own, since just by owning a low reference, the position of a thread can be detected by "low observers". As we will see soon, the threads' security levels are used to reenforce security effects: the writing effect is updated when a thread migrates, while the termination effect is updated when a remote access is attempted.

The Typing Judgments

As defined in Figure 4.6, the judgments of the type and effect system have the form:

$$\Sigma, \Gamma \vdash^{\check{m}_j}_{G,F} M : s, \tau$$

The meaning of Γ (the typing environment), M (the expression being typed), s (the security effect of M), G (the global flow policy) and F (flow policy of the

$$
\begin{array}{lrcl}
\textit{Thread Name Environment} & \Sigma & \subseteq & ((\boldsymbol{Nam} \cup \{?\}) \times 2^{\boldsymbol{Pri}}) \times (\check{\boldsymbol{Nam}} \times 2^{\boldsymbol{Pri}}) \\
\textit{where} & \Sigma\downarrow_{\boldsymbol{Nam}\times 2^{\boldsymbol{Pri}}} & : & (\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \to (\check{\boldsymbol{Nam}} \times 2^{\boldsymbol{Pri}}) \\
\textit{Typing Environments} & \Gamma & : & \boldsymbol{Var} \to \boldsymbol{Typ} \\
\textit{Typing Judgments} & & := & \Sigma;\Gamma \vdash^{\check{m}_j}_{G,F} M : s,\tau
\end{array}
$$

Figure 4.6: Syntax of Typing Judgments (see also Figure 4.1)

$$
[\textsc{Nil}]\ \Sigma;\Gamma \vdash () : \mathsf{unit} \qquad [\textsc{Flow}]\ \frac{\Sigma;\Gamma \vdash^{\check{m}_j}_{F\cup F'} M : s,\tau}{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (\text{flow } F' \text{ in } M) : s,\tau}
$$

$$
[\textsc{Abs}]\ \frac{\Sigma;\Gamma,x:\tau \vdash^{\check{m}_j}_{F} M : s,\sigma}{\Sigma;\Gamma \vdash (\lambda x.M) : \tau \xrightarrow[F,\check{m}_j]{s} \sigma} \qquad [\textsc{Rec}]\ \frac{\Sigma;\Gamma,x:\tau \vdash^{\check{m}_j}_{F} W : s,\tau}{\Sigma;\Gamma \vdash (\varrho x.W) : \tau}
$$

$$
[\textsc{BoolT}]\ \Sigma;\Gamma \vdash tt : \mathsf{bool} \qquad [\textsc{BoolF}]\ \Sigma;\Gamma \vdash ff : \mathsf{bool}
$$

$$
[\textsc{Var}]\ \Sigma;\Gamma,x:\tau \vdash x : \tau \qquad [\textsc{Loc}]\ \Sigma;\Gamma \vdash n_k.u_{l,\theta} : \theta\ \mathrm{ref}_{l,\Sigma(n_k)}
$$

$$
[\textsc{Ref}]\ \frac{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} M : s,\theta \qquad \begin{array}{c} j \preceq\ \ l \\ s.r, s.t \preceq_F l \end{array}}{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (\mathrm{ref}_{l,\theta}\ M) : s \curlyvee \langle \top, l, \top\rangle, \theta\ \mathrm{ref}_{l,\check{m}_j}}
$$

$$
[\textsc{Der}]\ \frac{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} M : s,\theta\ \mathrm{ref}_{l,\check{n}_k}}{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (?\ M) : s \curlyvee \langle l, \top, (\text{if } \check{m} \neq \check{n} \text{ then } j \curlyvee k \text{ else } \bot)\rangle, \theta}
$$

$$
[\textsc{Ass}]\ \frac{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} M : s,\theta\ \mathrm{ref}_{l,\check{n}_k} \quad \Sigma;\Gamma \vdash^{\check{m}_j}_{F} N : s',\theta \qquad \begin{array}{c} s.t \preceq_F s'.w \\ s.r, s'.r, s.t, s'.t, j \preceq_F l \end{array}}{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (M :=^{?} N) : s \curlyvee s' \curlyvee \langle \bot, l, (\text{if } \check{m} \neq \check{n} \text{ then } j \curlyvee k \text{ else } \bot)\rangle, \mathsf{unit}}
$$

$$
[\textsc{Cond}]\ \frac{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} M : s,\mathsf{bool} \qquad \begin{array}{c} \Sigma;\Gamma \vdash^{\check{m}_j}_{F} N_t : s_t,\tau \\ \Sigma;\Gamma \vdash^{\check{m}_j}_{F} N_f : s_f,\tau \end{array} \qquad s.r, s.t \preceq_F s_t.w, s_f.w}{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (\text{if } M \text{ then } N_t \text{ else } N_f) : s \curlyvee s_t \curlyvee s_f \curlyvee \langle \bot, \top, s.r\rangle, \tau}
$$

$$
[\textsc{App}]\ \frac{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} M : s,\tau \xrightarrow[F,\check{m}_j]{s'} \sigma \quad \Sigma;\Gamma \vdash^{\check{m}_j}_{F} N : s'',\tau \qquad \begin{array}{c} s.t \preceq_F s''.w \\ s.r, s''.r, s.t, s''.t \preceq_F s'.w \end{array}}{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (M\ N) : s \curlyvee s' \curlyvee s'' \curlyvee \langle \bot, \top, s.r \curlyvee s''.r\rangle, \sigma}
$$

$$
[\textsc{Seq}]\ \frac{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} M : s,\tau \quad \Sigma;\Gamma \vdash^{\check{m}_j}_{F} N : s',\sigma \quad s.t \preceq_F s'.w}{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (M;N) : s \curlyvee s',\sigma}
$$

$$
[\textsc{Thr}]\ \frac{j \preceq_F k \quad \check{n}\ \textit{fresh in}\ \Sigma \quad \Sigma,?_k : \check{n}_k;\Gamma \vdash^{\check{n}_k}_{\emptyset} M : s,\mathsf{unit}}{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (\mathrm{thread}_l\ M) : \langle \bot, s.w, \bot\rangle, \mathsf{unit}}
$$

$$
[\textsc{Mig}]\ \Sigma;\Gamma \vdash^{\check{m}_j}_{F} (\text{goto } d) : \langle \bot, j, \bot\rangle, \mathsf{unit}
$$

Figure 4.7: Type and Effect System

context, also assumed to contain G) is the same as in the previous chapter (see Section 3.4). The remaining parameters have the following meaning:

- Σ is a binary relation between decorated thread names extended with '$?_k$' (where '?' represents unknown thread names), and the set of decorated thread identifiers. We define $\mathrm{dom}(\Sigma)$ as $\{n_k \mid \exists \check{n}_k \,.\, (n_k, \check{n}_k) \in \Sigma\}$. In fact, the restriction of Σ to the domain $\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}$ (written $\Sigma \downarrow_{\boldsymbol{Nam}\times 2^{\boldsymbol{Pri}}}$) is assumed to be a function, where all thread names n are distinct. The only identifiers that are images of thread names are those that correspond to threads that have already created a reference – and whose name is the prefix of that address. The others are related to $?_k$ for some security level l, which represents the thread names that are created at runtime.

- The thread identifier $\check{m}_j$ identifies the thread to which the expression M belongs.

- The security level j represents a lower bound to the references that the thread owns and creates. It corresponds to the security level that is attributed to threads when they are created.

- The type τ is the type of the expression. The types we use in this chapter are similar to those of Chapter 3. The syntax (that can be seen in Figure 4.1) is repeated here:

$$\tau, \sigma, \theta \;\in\; \boldsymbol{Typ} \;::=\; t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta\ \mathrm{ref}_{l,\check{m}_j} \mid \tau \xrightarrow[G,\check{m}_j]{s} \sigma$$

  It includes annotations that are used to determine the *effects* of the expression that is being typed, as in the previous chapters. Similarly, to calculate them we take into account the level of the references that are accessed and the flow policy of the context – but, here we also distinguish between local and foreign references. For this purpose thread identifiers and security levels appear in the types as well.

In some of the typing rules we use the join operation on security effects:

Definition 4.4.1.

$$s \curlyvee_G s' \quad \stackrel{def}{\Leftrightarrow} \quad (s.r \curlyvee_G s'.r, s.w \curlywedge_G s'.w, s.t \curlyvee_G s'.t)$$

The type and effect system is given in Figure 4.7. Notice that it is syntax directed. We use some abbreviations: we write the flow relation with respect to the global flow policy as $\preceq$, meet $\curlywedge$ and join $\curlyvee$, instead of $\preceq_G$, $\curlywedge_G$ and $\curlyvee_G$, respectively; we also omit the global flow policy that appears as subscript of $\vdash^{\check{m}_j}_{G,F}$ and simply write $\vdash^{\check{m}_j}_{F}$; whenever we have $\forall F, \check{m}_j \,.\, \Sigma; \Gamma \vdash^{\check{m}_j}_{F} M : \langle \bot, \top, \bot \rangle, \tau$ we only write $\Sigma; \Gamma \vdash M : \tau$.

4.4.2 Typing Conditions

We must now convince ourselves that the type system indeed selects only safe threads, according to the Non-disclosure for Networks policy, defined in the previous section.

In rule Loc, since the name of the thread that owns the reference is given in the prefix, the corresponding thread identifier is found using Σ. In rule Ref, the reference that is created belongs to the thread identified by the superscript of the '$\vdash$'. We check that the security level that is declared for the new reference is greater than the level of the thread.

The body of an abstraction (rule Abs) is executed by the thread that applies it to an argument (see App), in the same flow context of that application. This is why the thread identifier and flow context of its execution are latent.

In rule Thr, a fresh identifier – image of an unknown thread name represented by '?' – is used to type the thread that is created. When a runtime thread is created by another runtime thread, the domain of Σ that is used to type the nested threads contains more than one entry using ?. The reason why the value of ? cannot be overwritten when typing nested thread creations is that we must keep a full record of the image of Σ, in order to guarantee that new thread identifiers that are attributed by the rule Thr are fresh. As we will see soon, these are used mainly to distinguish accesses to local references from accesses to foreign references (that are potentially remote).

Migration Leaks

The usual intuitions on treating termination leaks can be useful to understand the type system. In fact, suspension of a thread on an access to an absent reference can be seen as a non-terminating computation that can be unblocked by migration of concurrent threads. In other words, migration leaks are forms of termination leaks.

We have seen in Chapter 2 that termination leaks appear when a change to the low memory depends on the termination of a computation that precedes it, which in turn depends on high information. Example 4.11 shows how high information can leak due to a thread (n), in another domain, different from the one that performs the low assignment (m_1 and m_2). The key point in this example is that the synchronization between the two threads is made via the migration of n to a domain where m_1 or m_2 is located, at a time where the low assignments that are bound to occur in m_1 and m_2 are blocked by a suspension on an access to one of n's references.

From the point of view of n, it is not possible to know whether, in the domains it might migrate to, there are threads that are suspended on its arrival. But, as long as all the other threads are typable, one can ensure (see below) that the blocked low assignments are not lower than the accesses that are causing the suspension (i.e. $k \preceq L$). Then, the worst case assumed for n, the writing effect of the migration is updated with the security level of n, a lower bound k to the level of all its references. Notice that as a consequence, the rule Cond rejects thread n in a standard manner, since $H \not\preceq L$. In rule Mig, by adding the security level of the thread to the write effect of the migration construct, we thus prevent migrations of threads owning low references from depending on high information.

From the point of view of m_1 and m_2, it is not possible to know whether the arrival of the thread (n) that will unblock their computations depends on high information or not. But, as long as n is typable, one can insure (as above) that the level of the information on which the migration of n depends is lower or equal to the level of n itself (i.e. $H \preceq k$). Therefore, in rules Der and Ass, the

termination effect is updated with the level of the thread that owns the foreign reference we want to access.

In Examples 4.13 and 4.14, the low assignment can only occur if the threads m and n are located in the same domain. Therefore, also the position of m might be leaked when the low assignment occurs. This accounts for updating the termination level of the assignment (Ass) and the dereference (Der) with the security level of m as well.

Ref. The condition $j \preceq k$ ensures that the references that are created by a thread respect the security level of the thread, i.e. that they are not lower (with respect to the global flow policy) than it.

Ass. The insecure program in Example 4.15 is rejected by the condition $j \preceq_G l$ in rule Ass. Similarly, the program in Example 4.12 is rejected if j_1 or $j_2 \not\preceq L$, to prevent revealing information about the positions of m_1 and m_2. Notice that, in the typing rule, for the cases where $m = n$ the condition is satisfied anyway due to the meaning of k. There is an implicit condition, $k \preceq l$, which is satisfied by assumption, since the thread n owns the reference $n.y_L$.

Thr. The condition $j \preceq_F l$ rejects the insecure program:

$$d[(\text{thread}_L\ M)^{m_H}] \tag{4.22}$$

The reason why this program is considered insecure is that the presence of the high thread m, which should only be "visible" at level H, is indicated at the level L, at which the created thread is apparent.

4.4.3 Properties of Typed Expressions

Meaning of Effects

Unlike the effects given by the type systems of the previous two chapters, here it is not true that the termination effect of a typable expression is always downward bounded by its reading effect. The reason for this is that here termination of an expression does not depend only on the existence of non-terminating loops – which depend on tested values –, but also on the possibility of suspension on accesses to foreign references – which depend on the relative position of threads in the network.

We check that the intuitive meaning of the effects is indeed captured by our type system.

Lemma 4.4.2 (Update of Effects)**.**

1. *If* $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[(?\ n_k.u_{l,\theta})] : s,\tau$ *then* $l \preceq s.r$*. Also, if* $m \neq n$*, then* $k \curlyvee j \preceq s.t$*.*
2. *If* $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[(n_k.u_{l,\theta} :=^?\ V)] : s,\tau$*, then* $s.w \preceq l$*. Also, if* $m \neq n$*, then* $k \curlyvee j \preceq s.t$*.*
3. *If* $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[(\text{ref}_{l,\theta}\ V)] : s,\tau$*, then* $s.w \preceq l$*.*
4. *If* $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[(\text{goto}\ d)] : s,\tau$*, then* $s.w \preceq j$*.*

Proof. By induction on the structure of E. □

Subject Reduction

In order to establish the soundness of the type system of Figure 4.7 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 [Wright & Felleisen, 1994] in detail.

We start by remarking that a value, and more generally a pseudo-value, has no effect, and that this is properly reflected in the type system. Moreover, the typing of a pseudo-value does not depend on the thread identifier or current flow policy:

Remark 4.4.3. *If $W \in \boldsymbol{Pse}$ and $\Sigma;\Gamma \vdash_F^{\check{m}_j} W : s,\tau$, then for all thread identifiers $\check{n}_k$ and flow policies F', we have that $\Sigma;\Gamma \vdash_{F'}^{\check{n}_k} W : \langle\bot,\top,\bot\rangle,\tau$.*

The following lemma establishes some standard weakening and strengthening properties:

Lemma 4.4.4.

1. *If $\Sigma;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$ and $x \notin \mathrm{dom}(\Gamma)$ then $\Sigma;\Gamma,x:\sigma \vdash_F^{\check{m}_j} M : s,\tau$.*
2. *If $\Sigma;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$ and $\check{n}$ fresh in Σ then $\Sigma,?_k:\check{n}_k;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$.*
3. *If $\Sigma;\Gamma,x:\sigma \vdash_F^{\check{m}_j} M : s,\tau$ and $x \notin \mathrm{fv}(M)$ then $\Sigma;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$.*
4. *If $\Sigma;\Gamma \vdash_G^{\check{m}_j} M : s,\tau$ then $\Sigma;\Gamma \vdash_{G\cup F}^{\check{m}_j} M : s,\tau$.*

Proof. By induction on the inference of the type judgment. □

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

Lemma 4.4.5 (Substitution)**.**
If $\Sigma;\Gamma,x:\sigma \vdash_F^{\check{m}_j} M : s,\tau$ and $\Sigma;\Gamma \vdash W : \sigma$ then $\Sigma;\Gamma \vdash_F^{\check{m}_j} \{x \mapsto W\}M : s,\tau$.

Proof. By induction on the inference of $\Sigma;\Gamma,x:\tau \vdash_F^{\check{m}_j} M : s,\sigma$, and by case analysis on the last rule used in this typing proof, using the previous lemma.

Nil. Here $\{x \mapsto W\}M = M$, and since $x \notin \mathrm{fv}(M)$ then by Lemma 4.4.4 we have $\Sigma;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$.

Var. If $M = x$ then $s = \langle\bot,\top,\bot\rangle$, $\sigma = \tau$ and $\{x \mapsto W\}M = W$. By Remark 4.4.3, we have $\Sigma;\Gamma \vdash_F^{\check{m}_j} W : \langle\bot,\top,\bot\rangle,\tau$. If $M \neq x$ then $\{x \mapsto W\}M = M$, where $x \notin \mathrm{fv}(M)$. Therefore, by Lemma 4.4.4, we have $\Sigma;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$.

Abs. Here $M = (\lambda y.\bar{M})$, and $\Sigma;\Gamma,x:\sigma,y:\bar{\tau} \vdash_{\bar{F}}^{\check{n}_k} \bar{M} : \bar{s},\bar{\sigma}$ where $\tau = \bar{\tau} \xrightarrow[\bar{F},\check{n}_k]{\bar{s}} \bar{\sigma}$. We can assume that $y \notin \mathrm{dom}(\Gamma,x:\sigma)$ (otherwise rename y). Therefore $\{x \mapsto W\}(\lambda y.\bar{M}) = (\lambda y.\{x \mapsto W\}\bar{M})$. By assumption and Lemma 4.4.4 we can write $\Sigma;\Gamma,y:\bar{\tau} \vdash W : \sigma$. By induction hypothesis, $\Sigma;\Gamma,y:\bar{\tau} \vdash_{\bar{F}}^{\check{n}_k} \{x \mapsto W\}\bar{M} : \bar{s},\bar{\sigma}$. Then, by Abs, $\Sigma;\Gamma \vdash (\lambda y.\{x \mapsto W\}\bar{M}) : \tau$, and in particular $\Sigma;\Gamma \vdash_F^{\check{m}_j} (\lambda y.\{x \mapsto W\}\bar{M}) : s,\tau$.

Rec. Here $M = (\varrho y.\bar{W})$, and $\Sigma;\Gamma, x:\sigma, y:\tau \vdash_{\bar{F}}^{\check{n}_k} \bar{W} : \bar{s}, \tau$. We can assume that $y \notin \mathrm{dom}(\Gamma, x:\sigma)$ (otherwise rename y). Therefore $\{x \mapsto W\}(\varrho y.\bar{W}) = (\varrho y.\{x \mapsto W\}\bar{W})$. By assumption and Lemma 4.4.4 we have $\Sigma;\Gamma, y:\tau \vdash W : \sigma$. By induction hypothesis, $\Sigma;\Gamma, y:\tau \vdash_{\bar{F}}^{\check{n}_k} \{x \mapsto W\}\bar{W} : \bar{s}, \tau$. Then, by Rec, $\Sigma;\Gamma \vdash (\varrho y.\{x \mapsto W\}\bar{W}) : \tau$, and in particular we have $\Sigma;\Gamma \vdash_F^{\check{m}_j} (\varrho y.\{x \mapsto W\})\bar{W} : s, \tau$.

Cond. Here $M = (\text{if } \bar{M} \text{ then } N_t \text{ else } N_f)$ and we have $\Sigma;\Gamma, x:\sigma \vdash_F^{\check{m}_j} \bar{M} : \bar{s}, \mathsf{bool}$, $\Sigma;\Gamma, x:\sigma \vdash_F^{\check{m}_j} N_t : s_t, \tau_1$ and $\Sigma;\Gamma, x:\sigma \vdash_F^{\check{m}_j} N_f : s_f, \tau_2$ with $\bar{s}.r, \bar{s}.t \preceq_F s_t.w, s_f.w$ and $s = \bar{s} \curlyvee s_t \curlyvee s_f \curlyvee \langle \bot, \top, s.r\rangle, \tau$. By induction hypothesis, $\Sigma;\Gamma, x:\sigma \vdash_F^{\check{m}_j} \{x \mapsto W\}\bar{M} : \bar{s}, \mathsf{bool}$, $\Sigma;\Gamma, x:\sigma \vdash_F^{\check{m}_j} \{x \mapsto W\}N_t : s_t, \tau_1$ and $\Sigma;\Gamma, x:\sigma \vdash_F^{\check{m}_j} \{x \mapsto W\}N_f : s_f, \tau_2$. Therefore, $\Sigma;\Gamma, x:\sigma \vdash_F^{\check{m}_j} (\text{if } \{x \mapsto W\}\bar{M} \text{ then } \{x \mapsto W\}N_t \text{ else } \{x \mapsto W\}N_f) : s, \tau$ by rule Cond.

Thr. Here $M = (\mathrm{thread}_k\ \bar{M})$ and for $\check{n}$ fresh in Σ, we have that $\Sigma, ?_k : \check{n}_k;\Gamma, x:\sigma \vdash_{\emptyset}^{\check{n}_k} \bar{M} : s, \tau$, with $\tau = \mathsf{unit}$ and $s = \langle \bot, s.w, \bot\rangle$. Using assumption and Lemma 4.4.4 we have $\Sigma, ?_k : \check{n}_k;\Gamma \vdash W : \sigma$. By induction hypothesis, then $\Sigma, ?_k : \check{n}_k;\Gamma \vdash_{\emptyset}^{\check{m}_j} \{x \mapsto W\}\bar{M} : s, \tau$. Therefore, by rule Thr, $\Sigma;\Gamma \vdash_F^{\check{m}_j} (\mathrm{thread}_k\ \{x \mapsto W\}\bar{M}) : s, \tau$.

Flow. Here $M = (\text{flow } \bar{F} \text{ in } \bar{M})$ and $\Sigma;\Gamma, x:\sigma \vdash_{F\cup\bar{F}}^{\check{m}_j} \bar{M} : s, \tau$. By induction hypothesis, $\Sigma;\Gamma \vdash_{F\cup\bar{F}}^{\check{m}_j} \{x \mapsto W\}\bar{M} : s, \tau$. Then, by Flow, $\Sigma;\Gamma \vdash_F^{\check{m}_j} (\text{flow } \bar{F} \text{ in } \{x \mapsto W\}\bar{M}) : s, \tau$.

The proofs for the cases Loc, Ref, BoolT, BoolF and Mig are analogous to the one for Nil, while the proofs for App, Seq, Der and Ass are analogous to the one for Cond.

□

Lemma 4.4.6 (Replacement)**.**
If $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[M] : s, \tau$ is a valid judgment, then the proof gives M a typing $\Sigma;\Gamma \vdash_{F\cup\lceil \mathrm{E}\rceil}^{\check{m}_j} M : \bar{s}, \bar{\tau}$ for some $\bar{s}$ and $\bar{\tau}$ such that $\bar{s}.r \preceq s.r$, $s.w \preceq \bar{s}.w$ and $\bar{s}.t \preceq s.t$. In this case, if $\Sigma;\Gamma \vdash_{F\cup\lceil \mathrm{E}\rceil}^{\check{m}_j} N : \bar{s}', \bar{\tau}$ with $\bar{s}'.r \preceq \bar{s}.r$, $\bar{s}.w \preceq \bar{s}'.w$ and $\bar{s}'.t \preceq \bar{s}.t$, then $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[N] : s', \tau$, for some s' such that $s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$.

Proof. By induction on the structure of E.

$\mathrm{E}[M] = (\text{if } \bar{\mathrm{E}}[M] \text{ then } N_t \text{ else } N_f)$. By Cond, we have $\Sigma;\Gamma \vdash_F^{\check{m}_j} \bar{\mathrm{E}}[M] : \bar{s}, \mathsf{bool}$, and $\Sigma;\Gamma \vdash_F^{\check{m}_j} N_t : s_t, \tau$, $\Sigma;\Gamma \vdash_F^{\check{m}_j} N_f : s_f, \tau$ with $\bar{s}.r, \bar{s}.t \preceq_F s_t.w, s_f.w$ and $s = \bar{s} \curlyvee s_t \curlyvee s_f \curlyvee \langle \bot, \top, \bar{s}.r\rangle$. By induction hypothesis, the proof gives M a typing $\Sigma;\Gamma \vdash_{\hat{F}}^{\check{m}_j} M : \hat{s}, \hat{\tau}$, for $\hat{F}, \hat{s}, \hat{\tau}$ with $\hat{F} = F \cup \lceil \bar{\mathrm{E}}\rceil$ and $\hat{s}.r \preceq \bar{s}.r$, $\bar{s}.w \preceq \hat{s}.w$ and $\hat{s}.t \preceq \bar{s}.t$. Therefore, $\hat{s}.r \preceq s.r$, $s.w \preceq \hat{s}.w$ and $\hat{s}.t \preceq s.t$.

Also by induction hypothesis, $\Sigma;\Gamma \vdash_{\hat{F}}^{\check{m}_j} \bar{\mathrm{E}}[N] : \bar{s}', \bar{\tau}$, for some $\bar{s}'$ such that $\bar{s}'.r \preceq \bar{s}.r$, $s.w \preceq \bar{s}'.w$ and $\bar{s}'.t \preceq \bar{s}.t$. Since $\bar{s}'.r, \bar{s}'.t \preceq_F s_t.w, s_f.w$, then, again by Cond, we have $\Sigma;\Gamma \vdash_F^{\check{m}_j} (\text{if } \bar{\mathrm{E}}[N] \text{ then } N_t \text{ else } N_f) : \bar{s}' \curlyvee s_t \curlyvee s_f \curlyvee \langle \bot, \top, \bar{s}'.r\rangle, \tau$. We conclude by noting that $\bar{s}'.r \preceq s.r$, $s.w \preceq \bar{s}'.w$, and $\bar{s}'.t \curlyvee \bar{s}'.r \preceq s.t$.

$\mathbf{E}[M] = (\textbf{flow } F' \textbf{ in } \bar{\mathrm{E}}[M])$. By FLOW, we have $\Sigma;\Gamma \vdash^{\check{m}_j}_{F\cup F'} \bar{\mathrm{E}}[M] : s,\tau$. By induction hypothesis, the proof gives M a typing $\Sigma;\Gamma \vdash^{\check{m}_j}_{\hat{F}} M : \hat{s},\hat{\tau}$, for $\hat{F},\hat{s},\hat{\tau}$ with $\hat{F} = F \cup F' \cup \lceil\bar{\mathrm{E}}\rceil$ and $\hat{s}.r \preceq s.r$, $s.w \preceq \hat{s}.w$ and $\hat{s}.t \preceq s.t$.

Also by induction hypothesis, $\Sigma;\Gamma \vdash^{\check{m}_j}_{\hat{F}} \bar{\mathrm{E}}[N] : s',\tau$, for some s' such that $s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$. Then, again by FLOW, we have $\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (\text{flow } F' \text{ in } \bar{\mathrm{E}}[N]) : s',\tau$.

The proofs for the cases $\mathrm{E}[M] = \lceil(\mathrm{E}[M] :=^? N)\rceil$, $\mathrm{E}[M] = \lceil(V :=^? \mathrm{E}[M])\rceil$, $\mathrm{E}[M] = \lceil(?\ \mathrm{E}[M])\rceil$, $\mathrm{E}[M] = \lceil(\mathrm{E}[M]\ N)\rceil$, $\mathrm{E}[M] = \lceil(V\ \mathrm{E}[M])\rceil$, $\mathrm{E}[M] = \lceil(\mathrm{E}[M]; N)\rceil$ and $\mathrm{E}[M] = \lceil\mathsf{ref}_{l,\theta}\mathrm{E}[M]\rceil$, are analogous to the one for $\mathrm{E}[M] = (\text{if } \bar{\mathrm{E}}[M] \text{ then } N_t \text{ else } N_f)$. □

Finally we prove Subject Reduction, which states that computation preserves the type of threads, and that as the effects of an expression are performed, the security effects of the thread "weaken". To prove it, we assume that the value contained in references of type θ in the memories that we are dealing with have indeed type θ. The differences regarding Subject Reduction for the previous chapter (Theorem 3.4.2) lie only in the treatment of thread names. In particular, we ensure that, when a thread is created, it is typable with respect to a fresh thread identifier, in an environment where Σ is updated accordingly.

Theorem 4.4.7 (Subject reduction)**.**
If for some $\Sigma,\Gamma,s,\tau,F,m_j$ we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M : s,\tau$ and $\langle M^{m_j},T,S\rangle \xrightarrow[F']{N^{n_k}} \langle M'^{m_j},T',S'\rangle$ where all $a_{l,\theta} \in \mathrm{dom}(S)$ satisfy $\Sigma;\Gamma \vdash S(a_{l,\theta}) : \theta$, then $\exists s'$ such that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M' : s',\tau$, where $s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$. Furthermore, we have that $\exists \check{n}, s''$ such that $\Sigma, ?_k : \check{n}_k;\Gamma \vdash^{\check{n}_k}_{\emptyset} N : s'', \mathsf{unit}$ where $\check{n}$ is fresh in Σ, and $s.w \preceq s''.w$.

Proof. Suppose that $M = \bar{\mathrm{E}}[\bar{M}]$ and $\langle \bar{M}^{m_j},T,S\rangle \xrightarrow[\bar{F}]{\bar{N}^{n_k}} \langle \bar{M}'^{m_j},\bar{T}',\bar{S}'\rangle$. We start by observing that this implies $F' = \bar{F} \cup \lceil\bar{\mathrm{E}}\rceil$, $M' = \bar{\mathrm{E}}[\bar{M}']$, $\bar{N}^{n_k} = N^{n_k}$ and $\langle \bar{T}',\bar{S}'\rangle = \langle T',S'\rangle$. We can assume, without loss of generality, that $\bar{M}$ is the smallest in the sense that there is no $\hat{\mathrm{E}},\hat{M},\hat{N}$ such that $\hat{\mathrm{E}} \neq []$ and $\hat{\mathrm{E}}[\hat{M}] = \bar{M}$ for which we can write $\langle \hat{M}^{m_j},T,S\rangle \xrightarrow[\hat{F}]{\hat{N}^{n_k}} \langle \hat{M}'^{m_j},T',S'\rangle$.

By Lemma 4.4.6, we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\bar{\mathrm{E}}\rceil} \bar{M} : \bar{s},\bar{\tau}$ in the proof of $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} \bar{\mathrm{E}}[\bar{M}] : s,\tau$, for some $\bar{s}$ and $\bar{\tau}$. We now proceed by case analysis on the transition $\langle \bar{M}^{m_j},T,S\rangle \xrightarrow[\bar{F}]{\bar{N}^{n_k}} \langle \bar{M}'^{m_j},T',S'\rangle$, and prove that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\bar{\mathrm{E}}\rceil} \bar{M}' : \bar{s}',\bar{\tau}$, for some $\bar{s}'$ such that $\bar{s}'.r \preceq \bar{s}.r$, $\bar{s}.w \preceq \bar{s}'.w$ and $\bar{s}'.t \preceq \bar{s}.t$.

$\bar{M} = ((\lambda x.\hat{M})\ V)$. Here we have $\bar{M}' = \{x \mapsto V\}\hat{M}$. By rule APP, we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\hat{\mathrm{E}}\rceil} (\lambda x.\hat{M}) : \hat{s},\hat{\tau} \xrightarrow[F\cup\lceil\hat{\mathrm{E}}\rceil,\Sigma(m_j)]{\hat{s}'} \hat{\sigma}$, $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\hat{\mathrm{E}}\rceil} V : \hat{s}'',\hat{\tau}$, where $\hat{s}'.r \preceq \bar{s}.r$, $\bar{s}.w \preceq \hat{s}'.w$ and $\hat{s}'.t \preceq \bar{s}.t$. By ABS, then $\Sigma;\Gamma,x : \hat{\tau} \vdash^{\Sigma(m_j)}_{F\cup\lceil\hat{\mathrm{E}}\rceil} \hat{M} : \hat{s}',\hat{\sigma}$, and by Remark 4.4.3 we have $\Sigma;\Gamma \vdash V : \hat{\tau}$. Therefore, by Lemma 4.4.5, we get $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\hat{\mathrm{E}}\rceil} \{x \mapsto V\}\hat{M} : \hat{s}',\hat{\sigma}$.

$\bar{M} = (\textbf{if } tt \textbf{ then } N_t \textbf{ else } N_f)$. Here we have $\bar{M}' = N_t$. By COND, we have that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\hat{\mathrm{E}}\rceil} N_t : s_t,\bar{\tau}$, where $s_t.r \preceq \bar{s}.r$, $\bar{s}.w \preceq s_t.w$ and $s_t.t \preceq \bar{s}.t$.

$\bar{M} = (\mathbf{ref}_{l,\theta}\ V)$. Here we have $\bar{M}' = m_j.u_{l,\theta}$ (for some $m_j.u$ fresh in S). By Loc, we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\hat{\mathrm{E}}\rceil} m_j.u : \langle\bot,\top,\bot\rangle,\theta\ \mathrm{ref}_{l,\Sigma(m_j)}$.

$\bar{M} = (?\ n_k.u_{l,\theta})$. Here we have $\bar{M}' = S(n_k.u_{l,\theta})$. By assumption, we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\hat{\mathrm{E}}\rceil} S(n_k.u_{l,\theta}) : \langle\bot,\top,\bot\rangle,\theta$.

$\bar{M} = (\mathbf{flow}\ F'\ \mathbf{in}\ V)$. Here we have $\bar{M}' = V$. By rule Flow, we have that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\hat{\mathrm{E}}\rceil\cup F'} V : s,\tau$. Therefore, by Remark 4.4.3, we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup\lceil\hat{\mathrm{E}}\rceil} V : \langle\bot,\top,\bot\rangle,\bar{\tau}$.

The proof for the case $\bar{M} = (\varrho x.W)$ is analogous to the one for $\bar{M} = ((\lambda x.\hat{M})\ V)$, while the proofs for the cases $\bar{M} = (\mathrm{if}\ ff\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f)$ and $\bar{M} = (V;\hat{M})$ are analogous to the one for $\bar{M} = (\mathrm{if}\ tt\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f)$, and the ones for $\bar{M} = (n_k.u_{l,\theta} :=^? V)$, $\bar{M} = (\mathrm{thread}_l\ \hat{M})$ and $\bar{M} = (\mathrm{goto}\ d)$ are analogous to the one for $\bar{M} = (\mathrm{ref}_{l,\theta}\ V)$

By Lemma 4.4.6, we can now conclude that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} \bar{\mathrm{E}}[\bar{M}'] : s',\tau$, for some s' such that $s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$.

Now, if $N^{n_k} \neq ()$ (N^{n_k} is created), then $\exists l,\hat{N} : M = \bar{\mathrm{E}}[(\mathrm{thread}_l\ \hat{N})]$ and $\bar{N} = \hat{N}$. By Lemma 4.4.6, we have $\Sigma;\Gamma \vdash^{\Sigma(n_k)}_{\hat{F}\cup\lceil\bar{\mathrm{E}}\rceil} (\mathrm{thread}_l\ \hat{N}) : \hat{s},\mathsf{unit}$ in the proof of $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} \bar{\mathrm{E}}[(\mathrm{thread}_l\ \hat{N})] : s,\tau$, for some $\hat{s}$, and $\hat{\tau}$. By Thr, for some $\check{n}$ fresh in Σ we have $\Sigma,?_k : \check{n}_k;\Gamma \vdash^{\check{n}_k}_{\emptyset} \hat{N} : \hat{s},\mathsf{unit}$, where $\hat{s} = \langle\bot,s.w,\bot\rangle$. □

An expression that is typable in the type system of Figure 4.7 is clearly typable in the (standard) type system that is obtained by ignoring the security effects. This means that we could also state the full Type Safety result, which besides Subject Reduction insures that typable expressions are never blocked unless they are values.

Syntactically High Expressions

The notion of syntactically high expression that was defined in the previous chapters is extended to this setting as well. Similarly to Definition 3.4.3, it is defined with respect to the current flow policy. Furthermore, it takes a new parameter – the decorated name of the thread of which the expression is part. In this case, the writing effect is intended to be a lower bound to the level of the references that the expression can create or assign to, and also to the level of the thread that the expression appears in, in case it contains migration instructions.

Definition 4.4.8 (Syntactically "High" Expressions)**.** *An expression M is* syntactically (F,l,m_j)-high *if there exists Σ,Γ,s,τ such that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M : s,\tau$ with $s.w \not\preceq_F l$. The expression M is a* syntactically (F,l,m_j)-high function *if there exists Σ,Γ,s,τ such that $\Sigma;\Gamma \vdash M : \tau \xrightarrow[F,\Sigma(m_j)]{s} \sigma$ with $s.w \not\preceq_F l$.*

We are now able to prove that syntactically high expressions have an operationally high behavior.

Lemma 4.4.9 (High Expressions)**.** *If M is a syntactically (F,l,m_j)-high expression, then M^{m_j} is an operationally (F,l)-high thread.*

Proof. We show that if M is syntactically (F, l, m_j)-high, that is if there exists Σ, Γ, s, τ such that $\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} M : s, \tau$ with $s.w \not\preceq_F l$, and $\langle M^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle M'^{m_j}, T', S'\rangle$ then $S' =^{F,l} S$. This is enough since, by Subject Reduction (Theorem 4.4.7), both M' is syntactically (F, l, m_j)-high and N is syntactically (F, l, n_k)-high. We proceed by cases on the proof of the transition $\langle M^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle M'^{m_j}, T', S'\rangle$. The lemma is trivial in all the cases where $\langle T, S\rangle = \langle T', S'\rangle$.

$M = \mathrm{E}[(a_{\bar{l},\bar{\theta}} :=^? V)]$. Here $S' = [a_{\bar{l},\bar{\theta}} := V]S$ and so $s.w \preceq \bar{l}$ by Lemma 4.4.2. This implies $\bar{l} \not\preceq_F l$, hence $S' =^{F,l} S$.

$M = \mathrm{E}[(\mathbf{goto}\ d)]$. Here $T' = [m_j := d]T$ and so $s.w \preceq j$ by Lemma 4.4.2. This implies $j \not\preceq_F l$, hence $T' =^{F,l} T$.

The proof of the case $M = \mathrm{E}[(\mathrm{ref}_{l,\theta}\ V)]$ is analogous to the proof for $M = \mathrm{E}[(a_{l,\theta} :=^? V)]$, while the proof for the case $M = \mathrm{E}[(\mathrm{thread}_l\ M_0)]$ is analogous to the one for $M = \mathrm{E}[(\mathrm{goto}\ d)]$. □

4.4.4 Soundness

In this section we present and explain the proof of soundness of the type system of Figure 4.7 with respect to the notion of security of Definition 4.3.8. Proofs for each intermediate result are preceded by their "Rationale", which shortly gives the intuition behind the proof. In particular, the conditions of the typing rules that are used in the proof are pointed out. Refer to Subsections 2.4.2 and 4.4.2 for examples that justify the need for those conditions.

We set to prove that, under any global flow policy G, all sets of threads P that are typable using the type system of Figure 4.7 (on page 80) satisfy Non-Disclosure for Networks, given by Definition 4.3.8 (on page 75). Informally, this means that, whatever the security level that is chosen to be "low" (here that security level will be denoted by '*low*'), the set P always presents the same behavior according to a *weak* bisimulation on low-equal states: if two continuations P_1 and P_2 of P are related, and if P_1 can perform an execution step over a certain state, then P_2 can perform the same low changes to any low-equal state in zero or one step, while the two resulting continuations are still related. It is useful to start by analyzing the behavior 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 thread 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).

The ability of a thread to compute depends on whether its position in the network is the same as that of the references that it needs to access. This

means that to guarantee that a step is performed by a thread in two different states one must assume that it does not suspend on an access to an absent reference. The following guaranteed-transition result holds for low-equal states where, if the thread is about to access a reference, then either the thread owns that reference, or both the thread and the reference have a low security level.

Lemma 4.4.10 (Guaranteed Transitions)**.** *Suppose that M is typable for Σ, $\Sigma(m_j)$, F, and that if $M = \mathrm{E}[(n_k.u_{l,\theta} :=^{?} V)]$ or $M = \mathrm{E}[(?\ n_k.u_{l,\theta})]$ then either $j \curlyvee k \preceq_F low$ or $n = m$.*

If $\langle {M^{m_j}}, T_1, S_1\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle {M_1'}^{m_j}, T_1', S_1'\rangle$ such that $\bar{n}_{\bar{k}}$ is fresh for T_2 if $\bar{n}_{\bar{k}} \in \mathrm{dom}(T_1' - T_1)$ and a is fresh for S_2 if $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$ and for some F' we have $\langle T_1, S_1\rangle =^{F\cup F', low} \langle T_2, S_2\rangle$, then there exist M_2', T_2' and S_2' such that $\langle {M^{m_j}}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle {M_2'}^{m_j}, T_2', S_2'\rangle$ with $\langle T_1', S_1'\rangle =^{F\cup F', low} \langle T_2', S_2'\rangle$.

Rationale. When a typable thread m_j is about to perform an assignment or dereference of a reference that belongs to a thread n_k, the execution of this operation depends on m_j and n_k being located at the same domain. By assuming that either both j and k are low, or m and n are the same thread, we can conclude that, in low-equal states, m_j and n_k have the same location. Therefore, if M performs a transition in some state, it is able to perform it in a low-equal state as well.
When a thread n_k is created by m_j, we use the condition $j \preceq_F k$ of rule THR to ensure that either k is high (and therefore its creation does not change the low state) or m_j is a low thread (therefore n_k is created at the same place in two low-equal memories).

Proof. By case analysis on the proof of $\langle {M^{m_j}}, T_1, S_1\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle {M_1'}^{m_j}, T_1', S_1'\rangle$. In most cases, this transition does not modify or depend on the state $\langle T_1, S_1\rangle$, and we may let $M_2' = M_1'$ and $\langle T_2', S_2'\rangle = \langle T_2, S_2\rangle$.

$\boldsymbol{M = \mathrm{E}[(\mathbf{ref}_{l,\theta}\ V)]}$**.** Here $M' = \mathrm{E}[m_j.u_{l,\theta}]$, $F = \lceil \mathrm{E} \rceil$, $N^{\bar{n}_{\bar{k}}} = ()$, $T_1' = T_1$ and $S_1' = S_1 \cup \{m_j.u_{l,\theta} \mapsto V\}$. Since $m_j.u$ is fresh for S_2, we also have that $\langle {M^{m_j}}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle {M_1'}^{m_j}, T_2, S_2' \cup \{m_j.u_{l,\theta} \mapsto V\}\rangle$.

$\boldsymbol{M = \mathrm{E}[(?\ n_k.u_{l,\theta})]}$**.** Here $M' = \mathrm{E}[S_1(n_k.u_{l,\theta})]$, $F = \lceil \mathrm{E} \rceil$, $N^{\bar{n}_{\bar{k}}} = ()$, and $\langle T_1', S_1'\rangle = \langle T_1, S_1\rangle$. We have $\langle {M^{m_j}}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle \mathrm{E}[S_2(a_{l,\theta})]^{m_j}, T_2, S_2\rangle$, because $T_1 =^{F\cup F', low} T_2$ and either:

- $\boldsymbol{m = n}$**.** In this case M^{m_j} cannot suspend.
- $\boldsymbol{m \neq n}$ **and** $\boldsymbol{j \curlyvee k \preceq_F low}$**.** In this case $T_1(m_j) = T_2(m_j)$ and $T_1(n_k) = T_2(n_k)$. Since $T_1(m_j) = T_1(n_k)$, then $T_2(m_j) = T_2(n_k)$.

In other words, also in T_2 the threads m_j and n_k are located in the same domain.

$M = \mathbf{E}[(n_k.u_{l,\theta} :=^? V)]$. then $M' = \mathrm{E}[()]$, $F = \lceil \mathrm{E} \rceil$, $N^{\bar{n}_{\bar{k}}} = ()$, $T_1' = T_1$ and $S_1' = [n_k.u_{l,\theta} := V]S_1$. Analogously to the previous case, in T_2 the threads m_j and n_k are located in the same domain, so $\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle \mathrm{E}[()]^{m_j}, T_2, n_k.u_{l,\theta} V S_2\rangle$.

$M = \mathbf{E}[(\mathbf{thread}_{\bar{k}}\ \bar{M})]$. Here $M' = \mathrm{E}[()]$, $F = \emptyset$, $N^{\bar{n}_{\bar{k}}} = \bar{M}^{\bar{n}_{\bar{k}}}$, $T_1' = T_1 \cup \{\bar{n}_{\bar{k}} \mapsto T_1(m_j)\}$, and $S_1' = S_1$. Since n is fresh for T_2, we have $\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle \mathrm{E}[()]^{m_j}, T_2 \cup \{\bar{n}_{\bar{k}} \mapsto T_2(m_j)\}, S_2\rangle$. Notice that $T_1 \cup \{\bar{n}_{\bar{k}} \mapsto T_1(m_j)\} =^{F\cup F', low} T_2 \cup \{\bar{n}_{\bar{k}} \mapsto T_2(m_j)\}$, because $T_1 =^{F\cup F', low} T_2$ and if $l \preceq_{F\cup F'} low$, then by the condition $j \preceq_{F\cup F'} l$ in rule THR also $j \preceq_{F\cup F'} low$, in which case $T_1(m_j) = T_2(m_j)$.

$M = \mathbf{E}[(\mathbf{goto}\ d')]$. Then $M' = \mathrm{E}[()]$, $F = \emptyset$, $N^{\bar{n}_{\bar{k}}} = ()$, $T_1' = [m_j := d']T_1$ and $S_1' = S_1$. We have $\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle \mathrm{E}[()]^{m_j}, [m_j := d']T_2, S_2\rangle$.

□

The hypotheses of the previous lemma are fulfilled when the termination effect is low:

Remark 4.4.11. *Suppose that $M = \mathrm{E}[(n_k.u_{l,\theta} :=^? V)]$ or $M = \mathrm{E}[(?\ n_k.u_{l,\theta})]$, and that for some Σ, m_j, F, s and τ we have that $\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} M : s, \tau$. Then, $s.t \preceq_F low$ implies that either $j \curlyvee k \preceq_F low$ or $n = m$.*

We aim at proving that any typable thread M^{m_j} that has a low-termination effect always presents the same behavior according to a *strong* bisimulation on low-equal states: if two continuations $M_1^{m_j}$ and $M_2^{m_j}$ of M^{m_j} are related, and if $M_1^{m_j}$ can perform an execution step over a certain state, then $M_2^{m_j}$ 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^{m_j} under low-equal states should have the same "length", and in particular they are either 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 4.4.10).

The definition of $\mathcal{T}^{m_j}_{G,F,low}$, abbreviated $\mathcal{T}^{m_j}_{F,low}$ when the global flow policy is G, is given in Figure 4.8. 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 $\mathcal{T}^{m_j}_{F,low}$ 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 4.4.13. *If for some m_j, F and low we have that $M_1\ \mathcal{T}^{m_j}_{F,low}\ M_2$ and $M_1 \in \mathbf{Val}$, then $M_2 \in \mathbf{Val}$.*

We have seen in Splitting Computations (Lemma 4.2.1) 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

Definition 4.4.12 ($\mathcal{T}^{m_j}_{F,low}$)**.**
We have that $M_1\ \mathcal{T}^{m_j}_{F,low}\ M_2$ *if* $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M_1 : s_1, \tau$ *and* $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M_2 : s_2, \tau$ *for some* Σ, Γ, s_1, s_2 *and* τ *with* $s_1.t \preceq_F low$ *and* $s_2.t \preceq_F low$ *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* $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$*, or*

Clause 4. $M_1 = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ *and* $M_2 = (\mathrm{ref}_{l,\theta}\ \bar{M}_2)$ *with* $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$*, and* $l \not\preceq_F low$*, or*

Clause 5. $M_1 = (?\ \bar{M}_1)$ *and* $M_2 = (?\ \bar{M}_2)$ *with* $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$*, or*

Clause 6. $M_1 = (\bar{M}_1 :=^{?} \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 :=^{?} \bar{N}_2)$ *with* $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$*, and* $\bar{N}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{N}_2$*, and* $\bar{M}_1, \bar{M}_2$ *both have type* $\theta\ \mathrm{ref}_{l,\check{n}_k}$ *for some* θ *and* l *such that* $l \not\preceq_F low$*, or*

Clause 7. $M_1 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_1)$ *and* $M_2 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_2)$ *with* $\bar{M}_1\ \mathcal{T}^{m_j}_{F\cup F',low}\ \bar{M}_2$*.*

Figure 4.8: The relation $\mathcal{T}^{m_j}_{F,low}$

$\mathcal{T}^{m_j}_{F,low}$ 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 4.4.14. *If there exist* Σ, Γ, s, τ *such that* $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} \mathrm{E}[(?\ a_{l,\theta})] : s, \tau$ *with* $s.t \preceq_F low$ *and* $l \not\preceq_{F\cup[\mathrm{E}]} low$*, then for any values* $V_0, V_1 \in$ ***Val*** *such that* $\Sigma;\Gamma \vdash V_i : \theta$ *we have* $\mathrm{E}[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}[V_1]$*.*

Rationale. 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 Cond); to determine the function of an application ($s.r$ in the termination effect of App); to determine the argument of an application ($s''.r$ in the termination effect of App) of an application. The relation $\mathcal{T}$ 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 '$s.r \preceq_F l$', where s is the security effect of the program that is performing the access, and l 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 ($s.r \preceq_F l$ in rule REF); to determine a reference that is being assigned to ($s.r \preceq_F l$ in rule ASS); to determine a value that is being assigned ($s'.r \preceq_F l$ in rule ASS).

Proof. By induction on the structure of E.

$\mathbf{E}[(?\ a_{l,\theta})] = (?\ a_{l,\theta})$. We have $V_0\ \mathcal{T}^{m_j}_{F,low}\ V_1$ by Clause 1.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{E_1}[(?\ a_{l,\theta})]\ M)$. By APP we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\bar{\tau} \xrightarrow[F,\Sigma(m_j)]{\bar{s}'} \bar{\sigma}$ with $\bar{s}.r \preceq s.t$. By Lemma 4.4.2, we have $l \preceq \bar{s}.r$. Therefore $l \preceq_F s.t$, which contradicts the assumption that both $s.t \preceq_F low$ and $l \not\preceq_{F\cup \mathrm{E}} low$ hold.

$\mathbf{E}[(?\ a_{l,\theta})] = (V\ \mathbf{E_1}[(?\ a_{l,\theta})])$. By rule APP we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}'',\bar{\tau}$ with $\bar{s}''.r \preceq s.t$. By Lemma 4.4.2, we have $l \preceq \bar{s}''.r$. Therefore $l \preceq_F s.t$, which contradicts the assumption that both $s.t \preceq_F low$ and $l \not\preceq_{F\cup \mathrm{E}} low$ hold.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{if}\ \mathbf{E_1}[(?\ a_{l,\theta})]\ \mathbf{then}\ M_t\ \mathbf{else}\ M_f)$. By COND we have that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \mathsf{bool}$ with $\bar{s}.r \preceq s.t$. By Lemma 4.4.2, we have $l \preceq \bar{s}.r$. Therefore $l \preceq_F s.t$, which contradicts the assumption that both $s.t \preceq_F low$ and $l \not\preceq_{F\cup \mathrm{E}} low$ hold.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{E_1}[(?\ a_{l,\theta})]; M)$. By SEQ we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\bar{\tau}$ with $\bar{s}.t \preceq_F s.t$. Therefore $\bar{s}.t \preceq_F low$, and since $l \not\preceq_{F\cup \mathrm{E}} low$ implies $l \not\preceq_{F\cup \mathrm{E}_1} low$, then by induction hypothesis we have $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. By Lemma 4.4.6 and Clause 3 we can conclude.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{ref}_{l',\theta'}\ \mathbf{E_1}[(?\ a_{l,\theta})])$. By rule REF we have that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\bar{\tau}$ with $\bar{s}.r = s.r \preceq_F l'$ and $\bar{s}.t = s.t$. Therefore $\bar{s}.t \preceq_F low$, and since $l \not\preceq_{F\cup \mathrm{E}} low$ implies $l \not\preceq_{F\cup \mathrm{E}_1} low$, then by induction hypothesis we have $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. By Lemma 4.4.2 we have $l \preceq s.r$, so $s.r \not\preceq_F low$. Therefore, $l' \not\preceq_F low$, and we conclude by Lemma 4.4.6 and Clause 4.

$\mathbf{E}[(?\ a_{l,\theta})] = (?\ \mathbf{E_1}[a_{l,\theta}])$. By rule DER we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\bar{\tau}$ with $\bar{s}.t \preceq_F s.t$. Therefore $\bar{s}.t \preceq_F low$, and since $l \not\preceq_{F\cup \mathrm{E}} low$ implies $l \not\preceq_{F\cup \mathrm{E}_1} low$, then by induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. We conclude by Lemma 4.4.6 and Clause 5.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{E_1}[(?\ a_{l,\theta})] :=^{?} M)$. By rule ASS we have that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[a_{l,\theta}] : \bar{s},\bar{\theta}\ \mathrm{ref}_{\bar{l},\bar{n}_k}$ with $\bar{s}.t \preceq_F s.t$ and $\bar{s}.r \preceq_F \bar{l}$. Therefore $\bar{s}.t \preceq_F low$, and since $l \not\preceq_{F\cup \mathrm{E}} low$ implies $l \not\preceq_{F\cup \mathrm{E}_1} low$, then by induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. On the other hand, by Clause 2 we have $M\ \mathcal{T}^{m_j}_{F,low}\ M$. By Lemma 4.4.2 we have $l \preceq \bar{s}.r$, so $l \preceq_F \bar{l}$. Then, we must have $\bar{l} \not\preceq_F low$, since otherwise $l \preceq_{F\cup \mathrm{E}} low$. Therefore, we conclude by Lemma 4.4.6 and Clause 6.

$\mathrm{E}[(?\ a_{l,\theta})] = (V :=^{?} \mathbf{E_1}[(?\ a_{l,\theta})])$. By rule Ass we have that $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} V : \bar{s},\bar{\theta}\ \mathrm{ref}_{\bar{l},\bar{n}_k}$, and $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[a_{l,\theta}] : \bar{s}',\theta$ with $\bar{s}'.t \preceq_F s.t$ and $\bar{s}'.r \preceq_F \bar{l}$. Therefore $\bar{s}'.t \preceq_F low$, and since $l \not\preceq_{F\cup \mathrm{E}} low$ implies $l \not\preceq_{F\cup \mathrm{E}_1} low$, then by induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. On the other hand, by Clause 2 we have $V\ \mathcal{T}^{m_j}_{F,low}\ V$. By Lemma 4.4.2 we have $l \preceq \bar{s}'.r$, so $l \preceq_F \bar{l}$. Then, we must have $\bar{l} \not\preceq_F low$, since otherwise $l \preceq_{F\cup \mathrm{E}} low$. We then conclude by Lemma 4.4.6 and Clause 6.

$\mathrm{E}[(?\ a_{l,\theta})] = (\mathbf{flow}\ F'\ \mathbf{in}\ \mathbf{E_1}[(?\ a_{l,\theta})])$. By rule Flow we have $\Sigma;\Gamma \vdash_{F\cup F'}^{\Sigma(m_j)} V : s,\tau$. By induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F\cup F',low}\ \mathrm{E}_1[V_1]$, so we conclude by Lemma 4.4.6 and Clause 7.

□

We can now prove that $\mathcal{T}^{m_j}_{F,low}$ behaves as a kind of "strong bisimulation":

Proposition 4.4.15 (Strong Bisimulation for Low-Terminating Threads)**.** *If we have $M_1\ \mathcal{T}^{m_j}_{F,low}\ M_2$ and $\langle {M_1}^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle {M_1'}^{m_j}, T_1', S_1'\rangle$, with $\langle T_1, S_1\rangle =^{F\cup F',low} \langle T_2, S_2\rangle$ such that n is fresh for T_2 if $\bar{n}_{\bar{k}} \in \mathrm{dom}(T_1' - T_1)$ and a is fresh for S_2 if $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$, then there exist T_2', M_2' and S_2' such that $\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle {M_2'}^{m_j}, T_2', S_2'\rangle$ with $M_1'\ \mathcal{T}^{m_j}_{F,low}\ M_2'$ and $\langle T_1', S_1'\rangle =^{F,low} \langle T_2', S_2'\rangle$.*

Rationale. If M_1 and M_2 are equal (related by $\mathcal{T}$ using Clause 2), then since they have a low termination effect we can use Guaranteed Transitions (Lemma 4.4.10) 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 4.2.1) we conclude they have performed a high dereference. In Lemma 4.4.14 we have seen that this implies that the resulting expressions are still in the $\mathcal{T}$ 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, then, as long as suspension cannot occur, so can M_2.
Suspension could only occur when the dereference or assignment operations are eminent (i.e. when all arguments have computed into values). However, the possibility of suspension on an access to some reference that belongs to n_k is excluded by the fact that the threads $M_1^{m_j}$ and $M_2^{m_j}$ are assumed to have low termination effect. In fact, since if $m \neq n$ the termination of the threads depends on levels j and k, then both j and k must be low, which implies that m_j and n_k have the same position in low-equal memories. This is guaranteed by the type system in rules Der and Ass, when the termination effect is updated with $j \curlyvee k$.

Proof. By induction on the definition of $\mathcal{T}^{m_j}_{F,low}$. In the following, we use Subject Reduction (Theorem 4.4.7) 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 m_j, *low* and F, is a requirement for being in the $\mathcal{T}^{m_j}_{F,low}$ relation.

Clause 1. This case is not possible.

Clause 2. Here $M_1 = M_2$. By Guaranteed Transitions (Lemma 4.4.10) there exist T_2', M_2' and S_2' such that $\langle M_2^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle M_2'^{m_j}, T_2', S_2'\rangle$ with $\langle T_1', S_1'\rangle =^{F\cup F', low} \langle T_2', S_2'\rangle$.

- $\boldsymbol{M_2' = M_1'}$**.** Then we have $M_1'\ \mathcal{T}^{m_j}_{F,low}\ M_2'$, by Clause 2 and Subject Reduction (Theorem 4.4.7).
- $\boldsymbol{M_2' \neq M_1'}$**.** Then by Splitting Computations (Lemma 4.2.1) we have that $(N^{\bar{n}_{\bar{k}}} = ())$ and there exists E and $a_{l,\theta}$ such that $F' = \lceil \mathrm{E} \rceil$, $M_1' = \mathrm{E}[S_1(a_{l,\theta})]$, $M_2' = \mathrm{E}[S_2(a_{l,\theta})]$, $\langle T_1', S_1'\rangle = \langle T_1, S_1\rangle$ and $\langle T_2', S_2'\rangle = \langle T_2, S_2\rangle$. Since $S_1(a_{l,\theta}) \neq S_2(a_{l,\theta})$, we have $l \not\preceq_{F\cup F'} low$. Therefore, $M_1'\ \mathcal{T}^{m_j}_{F,low}\ M_2'$, by Lemma 4.4.14 above.

Clause 3. Here $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ where $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$. Then either:

- $\boldsymbol{\bar{M}_1}$ **can compute.** In this case $M_1' = (\bar{M}_1'; \bar{N})$ with $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use the induction hypothesis, Clause 3 and Subject Reduction (Theorem 4.4.7) to conclude.
- $\boldsymbol{\bar{M}_1}$ **is a value.** In this case $M_1' = \bar{N}$ and $F' = \emptyset$, $N^{\bar{n}_{\bar{k}}} = ()$ and $\langle T_1', S_1'\rangle = \langle T_1, S_1\rangle$. We have $\bar{M}_2 \in \boldsymbol{Val}$ by Remark 4.4.13, hence $\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{N}^{m_j}, T_2, S_2\rangle$, and we conclude using Clause 2 and Subject Reduction (Theorem 4.4.7).

Clause 4. Here $M_1 = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ and $M_2 = (\mathrm{ref}_{l,\theta}\ \bar{M}_2)$ where $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$, and $l \not\preceq_F low$. There are two cases.

- $\boldsymbol{\bar{M}_1}$ **can compute.** In this case $M_1' = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ with $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use the induction hypothesis, Subject Reduction (Theorem 4.4.7) and Clause 4 to conclude.
- $\boldsymbol{\bar{M}_1}$ **is a value.** In this case $M_1' = a_{l,\theta}$, with a fresh for S_1, $F' = \emptyset$, $N^{\bar{n}_{\bar{k}}} = ()$ and $\langle T_1', S_1'\rangle = \langle T_2, S_1 \cup \{a_{l,\theta} \mapsto \bar{M}_1\}\rangle$ (and therefore a is also fresh for S_2). Then $\bar{M}_2 \in \boldsymbol{Val}$ by Remark 4.4.13, and therefore $\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle {a_{l,\theta}}^{m_j}, T_2', S_2 \cup \{a_{l,\theta} \mapsto \bar{M}_2\}\rangle$. If we let $S_2' = S_2 \cup \{a_{l,\theta} \mapsto \bar{M}_2\}$ then $\langle T_1', S_1'\rangle =^{F,low} \langle T_2', S_2'\rangle$ since $l \not\preceq_F low$. We conclude using Clause 1 and Subject Reduction (Theorem 4.4.7).

Clause 5. Here $M_1 = (?\ \bar{M}_1)$ and $M_2 = (?\ \bar{M}_2)$ where $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$. We distinguish two sub-cases.

- $\boldsymbol{\bar{M}_1}$ **can compute.** In this case $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use the induction hypothesis, Subject Reduction (Theorem 4.4.7) and Clause 5 to conclude.

$\bar{M}_1$ is a value. Then $\bar{M}_1 = n_k.u_{l,\theta}$ and $M_1' \in \boldsymbol{Val}$, $\langle T_1', S_1' \rangle = \langle T_1, S_1 \rangle$, $F' = \emptyset$ and $N^{\bar{n}_{\bar{k}}} = ()$. By Remark 4.4.13, $\bar{M}_2 \in \boldsymbol{Val}$, and since M_1 and M_2 have the same type, it must be a reference $n_k.v_{l',\theta}$. Notice also that $T_1(n_k) = T_1(m_j)$.

- ***n* ≠ *m*.** Then, by rule DER, we have $j \curlyvee k \preceq s.t$, and therefore $j \curlyvee k \preceq_{F \cup F'} low$. Since $T_1 =^{F \cup F', low} T_2$, then $T_1(m_j) = T_2(m_j)$ and $T_1(n_k) = T_2(n_k)$.
- ***n* = *m*.** Then it is immediate that $T_2(m_j) = T_2(n_k)$.

In both the above cases, $T_2(n_k) = T_2(m_j)$, and so $\langle {M_2}^{m_j}, T_2, S_2 \rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle {M_2}'^{m_j}, T_2, S_2 \rangle$ with ${M_2}' \in \boldsymbol{Val}$. We then conclude using Clause 1 and Subject Reduction (Theorem 4.4.7).

Clause 6. Here we have $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ where $\bar{M}_1 \ \mathcal{T}^{m_j}_{F,low} \ \bar{M}_2$, $\bar{N}_1 \ \mathcal{T}^{m_j}_{F,low} \ \bar{N}_2$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta \ \mathrm{ref}_{l,\bar{n}_k}$ for some θ and l such that $l \not\preceq_F low$. We distinguish three sub-cases.

$\bar{M}_1$ can compute. In this case $\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle$. We use the induction hypothesis, Subject Reduction (Theorem 4.4.7) and Clause 6 to conclude.

$\bar{M}_1$ is value, but $\bar{N}_1$ can compute. In this case we have $\langle \bar{N}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{N}_1'^{m_j}, T_1', S_1' \rangle$. By Remark 4.4.13 also $\bar{M}_2 \in \boldsymbol{Val}$. We use the induction hypothesis, Subject Reduction (Theorem 4.4.7) and Clause 6 to conclude.

$\bar{M}_1$ and $\bar{M}_1$ are values. Then $\bar{M}_1 = n_k.u_{l,\theta}$ and $M_1' = ()$, $\langle T_1', S_1' \rangle = \langle T_1, \{V \mapsto \bar{M}_1\} S_1 \rangle$, $F' = \emptyset$ and $N^{\bar{n}_{\bar{k}}} = ()$. By Remark 4.4.13, also $\bar{M}_2, \bar{N}_2 \in \boldsymbol{Val}$, and since $\bar{M}_1$ and $\bar{M}_2$ have the same type, $\bar{M}_2$ must be a reference $n_k.v_{l',\theta'}$. Notice that $T_1(n_k) = T_1(m_j)$.

- ***n* ≠ *m*.** Then, by ASS, we have $j \curlyvee k \preceq s.t$, therefore $j \curlyvee k \preceq_{F \cup F'} low$. Since $T_1 =^{F \cup F', low} T_2$, then $T_1(m_j) = T_2(m_j)$ and $T_1(n_k) = T_2(n_k)$.
- ***n* = *m*.** Then it is immediate that $T_2(m_j) = T_2(n_k)$.

In both the above cases, $T_2(n_k) = T_2(m_j)$, and so $\langle {M_2}^{m_j}, T_2, S_2 \rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle {M_2}'^{m_j}, T_2, \{V' \mapsto \bar{M}_2\} S_2 \rangle$ with $\bar{M}_2' \in \boldsymbol{Val}$. Since $l \not\preceq_F low$, then $\{V \mapsto \bar{M}_1\} S_1 =^{F \cup F', low} \{V' \mapsto \bar{M}_2\} S_2$. We then conclude using Clause 1 and Subject Reduction (Theorem 4.4.7).

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

$\bar{M}_1$ can compute. In this case $\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F'']{N^{\bar{n}_{\bar{k}}}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle$ with $F' = \bar{F} \cup F''$. By induction hypothesis, we have $\langle \bar{M}_2^{m_j}, T_2, S_2 \rangle \xrightarrow[F'']{N^{\bar{n}_{\bar{k}}}} \langle \bar{M}_2'^{m_j}, T_2', S_2' \rangle$, and $M_1' \ \mathcal{T}^{m_j}_{F \cup \bar{F}, low} \ M_2'$ and $\langle T_1', S_1' \rangle =^{F \cup \bar{F}, low} \langle T_2', S_2' \rangle$. Notice that $\langle T_1', S_1' \rangle =^{F, low} \langle T_2', S_2' \rangle$. We use Subject Reduction (Theorem 4.4.7) and Clause 4 to conclude.

$\bar{M}_1$ is a value. In this case $M'_1 = \bar{M}_1$, $F' = \emptyset$, $N^{\bar{n}_{\bar{k}}} = ()$ and $\langle T'_1, S'_1\rangle = \langle T_1, S_1\rangle$. Then $\bar{M}_2 \in \boldsymbol{Val}$ by Remark 4.4.13, and so $\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{M}_2^{m_j}, T_2, S_2\rangle$. We conclude using Clause 1 and Subject Reduction (Theorem 4.4.7).

□

We have seen in Remark 4.4.13 that when two expressions are related by $\mathcal{T}^{m_j}_{F,low}$ 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 $\mathcal{T}^{m_j}_{F,low}$ 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 4.4.16 (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 $M' \in \boldsymbol{Val}$.*

Lemma 4.4.17. *If for some m_j, F and low we have that $M\ \mathcal{T}^{m_j}_{F,low}\ N$ and $M\dagger$, then $N\dagger$.*

Rationale. 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 Threads (Proposition 4.4.15), 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 $\neg N\dagger$. That means that there exists a finite number of states $\langle T_1, S_1\rangle, \ldots, \langle T_n, S_n\rangle$ and $\langle T'_1, S'_1\rangle, \ldots, \langle T'_n, S'_n\rangle$ and of expressions N_1, $\ldots$, N_n such that

$$\begin{array}{rcl} \langle N, T_1, S_1\rangle & \rightarrow & \langle N_1, T'_1, S'_1\rangle \text{ and} \\ \langle N_1, T_2, S_2\rangle & \rightarrow & \langle N_2, T'_2, S'_2\rangle \text{ and} \\ & \vdots & \\ \langle N_{n-1}, T_n, S_n\rangle & \rightarrow & \langle N_n, T'_n, S'_n\rangle \end{array}$$

and such that $N_n \in \boldsymbol{Val}$. By Strong Bisimulation for Low-Terminating Threads (Proposition 4.4.15), we have that there exists a finite number of states $\langle \bar{T}'_1, \bar{S}'_1\rangle$, $\ldots$, $\langle \bar{T}'_n, \bar{S}'_n\rangle$ and of expressions $\bar{M}_1$, $\ldots$, $\bar{M}_n$ such that

$$\begin{array}{rcl} \langle M, T_1, S_1\rangle & \rightarrow & \langle M_1, \bar{T}'_1, \bar{S}'_1\rangle \text{ and} \\ \langle M_1, T_2, S_2\rangle & \rightarrow & \langle M_2, \bar{T}'_2, \bar{S}'_2\rangle \text{ and} \\ & \vdots & \\ \langle M_{n-1}, T_n, S_n\rangle & \rightarrow & \langle M_n, \bar{T}'_n, \bar{S}'_n\rangle \end{array}$$

such that:

$$M_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{N}_1,\ \textit{and}\ \ldots,\ \textit{and}\ M_n\ \mathcal{T}^{m_j}_{F,low}\ \bar{N}_n$$

By Remark 4.4.13, we then have that $M_n \in \boldsymbol{Val}$. Since M_n is a derivative of M, we conclude that $\neg M\dagger$. □

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

Lemma 4.4.18 (Composition of High Expressions)**.** *Suppose that M^{m_j} is typable in Σ and F. Then:*

1. *If $M = (M_1\ M_2)$ and M_1 is a syntactically (F, low, m_j)-high function and either*
   - *$M_1\dagger$ and ${M_1}^{m_j} \in \mathcal{H}_{F,low}$, or*
   - *${M_1}^{m_j}, {M_2}^{m_j} \in \mathcal{H}_{F,low}$,*

   then $M^{m_j} \in \mathcal{H}_{F,low}$.
2. *If $M = (\text{if } M_1 \text{ then } M_t \text{ else } M_f)$ and ${M_1}^{m_j}, {M_t}^{m_j}, {M_f}^{m_j} \in \mathcal{H}_{F,low}$, then $M^{m_j} \in \mathcal{H}_{F,low}$.*
3. *If $M = (\text{ref}_{l,\theta}\ M_1)$ and $l \not\preceq_F low$ and ${M_1}^{m_j} \in \mathcal{H}_{F,low}$, then $M^{m_j} \in \mathcal{H}_{F,low}$.*
4. *If $M = (M_1; M_2)$ and either*
   - *$M_1\dagger$ and ${M_1}^{m_j} \in \mathcal{H}_{F,low}$, or*
   - *${M_1}^{m_j}, {M_2}^{m_j} \in \mathcal{H}_{F,low}$,*

   then $M^{m_j} \in \mathcal{H}_{F,low}$.
5. *If $M = (M_1 :=^? M_2)$ and M_1 has type $\theta\ \text{ref}_{l,\check{n}_k}$ with $l \not\preceq_F low$ and either*
   - *$M_1\dagger$ and ${M_1}^{m_j} \in \mathcal{H}_{F,low}$, or*
   - *${M_1}^{m_j}, {M_2}^{m_j} \in \mathcal{H}_{F,low}$,*

   then $M^{m_j} \in \mathcal{H}_{F,low}$.
6. *If $M = (\text{flow } F' \text{ in } M_1)$ and ${M_1}^{m_j} \in \mathcal{H}_{F\cup F',low}$, then $M^{m_j} \in \mathcal{H}_{F,low}$.*

Rationale. 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, low, m_j)-high function. The other cases are analogous or simpler.

$M_1\dagger$ and ${M_1}^{m_j} \in \mathcal{H}_{F,low}$. Let $\mathcal{F}$ be a set of threads that includes $\mathcal{H}_{F,low}$, and that contains the threads $(M_1\ M_2)^{m_j}$ provided that they are typable in F, and satisfy $M_1 \notin \textbf{\textit{Val}}$ and $M_1^{m_j} \in \mathcal{F}$ and M_1 is a (F, low, m_j)-high function. Assume that an application $M = (M_1\ M_2)$ such that $M_1\dagger$ and ${M_1}^{m_j} \in \mathcal{H}_{F,low}$ performs the transition $\langle {M}^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle {M'}^{m_j}, T', S'\rangle$. We show that this implies ${M'}^{m_j}, N^{n_k} \in \mathcal{F}$ and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$.

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 $\langle {M_1}^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle {M_1'}^{m_j}, T', S'\rangle$. Since ${M_1}^{m_j} \in \mathcal{H}_{F,low}$, then also ${M_1'}^{m_j}, N^{n_k} \in \mathcal{H}_{F,low}$, thus ${M_1'}^{m_j}, N^{n_k} \in \mathcal{F}$, and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$. By Subject Reduction (Theorem 4.4.7), M_1' is a (F, low)-high function, and since $M_1\dagger$ then $M_1' \notin \textbf{\textit{Val}}$. Hence ${M'}^{m_j} \in \mathcal{F}$.

${M_1}^{m_j}, {M_2}^{m_j} \in \mathcal{H}_{F,low}$. Let $\mathcal{F}$ be a set of pools of threads that includes $\mathcal{H}_{F,low}$, and that contains threads $(M_1\ M_2)^{m_j}$ provided they are typable in F and satisfy $M_1^{m_j}, M_2^{m_j} \in \mathcal{F}$ and M_1 is a (F, low, m_j)-high function. Assume that such an application $M = (M_1\ M_2)$ performs the transition $\langle {M}^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle {M'}^{m_j}, T', S'\rangle$. We show that this implies ${M'}^{m_j}, N^{n_k} \in \mathcal{F}$ and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$.

- **M_1 and M_2 are values.** Then $M_1 = (\lambda x.\bar{M}_1)$, $M' = \{x \mapsto M_2\}\bar{M}_1$ and $N' = \langle\rangle$, $\langle T', S'\rangle = \langle T, S\rangle$. Since M_1 is a (F, low, m_j)-high function, then by Abs $\bar{M}_1$ is syntactically (F, low, m_j)-high, and by Substitution (Lemma 4.4.5), also M' is syntactically (F, low, m_j)-high. Therefore, by High Expressions (Lemma 4.4.9), ${M'}^{m_j} \in \mathcal{H}_{F,low}$.
- **M_1 can compute.** Then we have $M' = (M_1'\ M_2)$ with $\langle {M_1}^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle {M_1'}^{m_j}, T', S'\rangle$. Since ${M_1}^{m_j} \in \mathcal{H}_{F,low}$, then also ${M_1'}^{m_j}, N^{n_k} \in \mathcal{F}$ and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$. By Subject Reduction (Theorem 4.4.7) M_1' is a (F, 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 $\langle {M_2}^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle {M_2'}^{m_j}, T', S'\rangle$. Since ${M_2}^{m_j}, N^{n_k} \in \mathcal{H}_{F,low}$, then also ${M_2'}^{m_j}, N^{n_k} \in \mathcal{F}$ and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$. Hence $M' \in \mathcal{F}$.

□

Lemma 4.4.19. *If for some m_j, F and low we have that $M_1\ \mathcal{T}^{m_j}_{F,low}\ M_2$ and $M_1 \in \mathcal{H}_{F,low}$, then $M_2 \in \mathcal{H}_{F,low}$.*

Rationale. The proof relies on the fact that if an expression M_1 of the form $(\bar{M_1}; \bar{N_1})$ or $(\bar{M_1} := \bar{N_1})$ is operationally high, in spite of $\bar{N_1}$ not being operationally high, then $\bar{M_1}$ is non-resolvable. To see this, note that if M_1 were not non-resolvable, we would have, for some value V, that $(V; \bar{N_1})$ or $(V := \bar{N_1})$ would be derivatives of M_1. We can then see that, for all the computation steps that can be performed by any of $\bar{N_1}$'s derivatives, there is a corresponding one that can be performed by a derivative of M_1. Since $\bar{N_1}$ 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 $\bar{N_1}$ to be operationally high. However, by the above observation and by Lemma 4.4.17 this implies that $\bar{M_2}$ is non-resolvable. We can then argue that the expressions in the $\mathcal{T}$ relation have the same "composition", and conclude that they are operationally high using Composition of High Expressions (Lemma 4.4.18).

Proof. By induction on the definition of $M_1 \; \mathcal{T}^{m_j}_{F,low} \; M_2$.

Clause 1. Direct.

Clause 2. Direct.

Clause 3. Here $M_1 = (\bar{M_1}; \bar{N})$ and $M_2 = (\bar{M_2}; \bar{N})$ with $\bar{M_1} \; \mathcal{T}^{m_j}_{F,low} \; \bar{M_2}$. Clearly we have that $\bar{M_1} \in \mathcal{H}_{F,low}$, so by induction hypothesis, also $\bar{M_2} \in \mathcal{H}_{F,low}$. We distinguish two sub-cases:

- $\boldsymbol{\bar{N} \in \mathcal{H}_{F,low}}$**.** Then, $\bar{M_2}, \bar{N} \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.
- $\boldsymbol{\bar{N} \notin \mathcal{H}_{F,low}}$**.** Then $\bar{M_1}\dagger$, and by Lemma 4.4.17 also $\bar{M_2}\dagger$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

Clause 4. Here $M_1 = (\mathrm{ref}_{l,\theta} \; \bar{M_1})$ and $M_2 = (\mathrm{ref}_{l,\theta} \; \bar{M_2})$ where $\bar{M_1} \; \mathcal{T}^{m_j}_{F,low} \; \bar{M_2}$, and $l \not\preceq_F low$. Clearly we have that $\bar{M_1} \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M_2} \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

Clause 5. Here $M_1 = (?\ \bar{M_1})$ and $M_2 = (?\ \bar{M_2})$ where $\bar{M_1} \; \mathcal{T}^{m_j}_{F,low} \; \bar{M_2}$. Clearly we have that $\bar{M_1} \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M_2} \in \mathcal{H}_{F,low}$. This implies that $\bar{M_2} \in \mathcal{H}_{F,low}$.

Clause 6. Here we have $M_1 = (\bar{M_1} :=^{?} \bar{N_1})$ and $M_2 = (\bar{M_2} :=^{?} \bar{N_2})$ where $\bar{M_1} \; \mathcal{T}^{m_j}_{F,low} \; \bar{M_2}$, and $\bar{M_1}, \bar{M_2}$ both have type $\theta \; \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$, and $\bar{N_1} \; \mathcal{T}^{m_j}_{F,low} \; \bar{N_2}$. Clearly we have that $\bar{M_1} \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M_2} \in \mathcal{H}_{F,low}$. We distinguish two sub-cases:

- $\boldsymbol{\bar{N_2} \in \mathcal{H}_{F,low}}$**.** Then, $\bar{M_2}, \bar{N_2} \in \mathcal{H}_{F,low}$ where $\bar{M_2}$ has type $\theta \; \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

$\bar{N}_2 \notin \mathcal{H}_{F,low}$. Then $\bar{M}_1\dagger$, and by Lemma 4.4.17 also $\bar{M}_2\dagger$. Therefore, since $\bar{M}_2$ has type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

Clause 7. Here we have $M_1 = (\mathrm{flow}\ F' \ \mathrm{in}\ \bar{M}_1)$ and $M_2 = (\mathrm{flow}\ F' \ \mathrm{in}\ \bar{M}_2)$ with $\bar{M}_1\ \mathcal{T}^{m_j}_{F\cup F',low}\ \bar{M}_2$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M}_2 \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

□

Behavior of Typable Low Expressions

In this second phase of the proof, we consider the general case of threads 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 4.3.9 and 4.4.8).

Since we are considering the general case where threads do not necessarily have a low termination effect we cannot, as we did in the previous section, state a guaranteed-transition result. However, from Guaranteed Transitions (Lemma 4.4.10) and Remark 4.4.11 we can guarantee transitions in the cases $M \neq \mathrm{E}[(n_k.u_{l,\theta} :=^? V)]$ and $M \neq \mathrm{E}[(?\ n_k.u_{l,\theta})]$, as well as for these two cases when M is low-terminating. The following lemma covers the remaining cases by asserting that if $M = \mathrm{E}[(?\ a_{l,\theta})]$ when M is not low-terminating, then M is operationally high (therefore it cannot perform low changes on the state).

Lemma 4.4.20 (Potentially Suspensive Transitions)**.** *Suppose that M^{m_j} is typable in Σ and F, and consider the two cases where $M = \mathrm{E}[(n_k.u_{l,\theta} :=^? V)]$ and $M = \mathrm{E}[(?\ n_k.u_{l,\theta})]$ with $j \curlyvee k \not\preceq_F low$ and $n \neq m$. Then $M^{m_j} \in \mathcal{H}_{F,low}$.*

Rationale. By Remark 4.4.11, the assumptions $j \curlyvee k \not\preceq_F low$ and $n \neq m$ indicate that, as far as the type system is concerned, accesses performed by a thread m_j to a reference that belongs to a thread n_k under low-equal memories are potentially suspensive. This means that, according to the principle that no low-writes can follow high-terminating portions of the program, then a thread that is performing such an access must be high.
The above mentioned principle is guaranteed by the type system using conditions of the form '$s.t \preceq_F$' on the effects and security levels representing writes that are to be performed after the execution of a subprogram with security effect s. More precisely, conditions are imposed when the foreign access is used: to create a reference ($s.t \preceq_F l$ in rule Ref); to determine a reference that is being assigned to ($s.t \preceq_F s'.w$ and $s.t \preceq_F l$ in rule Ass); to determine a value that is being assigned ($s'.t \preceq_F l$ in rule Ass); to determine the predicate of a conditional ($s.t \preceq_F s_t.w, s_f.w$ in rule Cond); to determine a function that is being applied ($s.t \preceq_F s'.w$ and $s.t \preceq_F s''.w$ in

rule App); to determine an argument to which a function 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).

Proof. By induction on the structure of E. Consider $M = \mathrm{E}[M_0]$, where either $M_0 = (n_k.u_{l,\theta} :=^? V)$ or $M_0 = (?\ n_k.u_{l,\theta})$.

$\mathbf{E}[M_0] = M_0$. Direct.

$\mathbf{E}[M_0] = (\mathbf{E_1}[M_0]\ M_1)$. Then by rule App we have that $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \tau_1 \xrightarrow[F,\check{m}_j]{s_1'} \sigma_1$ and $\Sigma;\Gamma \vdash_F^{\check{m}_j} M_1 : s_1'', \tau_1$ with $s_1.t \preceq_F s_1''.w$ and $s_1.t \preceq_F s_1'.w$. By Remark 4.4.11 we have $s_1.t \not\preceq_F low$, and so $s_1''.w \not\preceq_F low$. Therefore, $s_1'.w \not\preceq_F low$, and $s_1''.w \not\preceq_F low$, which means that $\mathrm{E}_1[M_0]$ is a syntactically (F, low, m_j)-high function and M_1 is (F, low, m_j)-high. By High Expressions (Lemma 4.4.9) we have ${M_1}^{m_j} \in \mathcal{H}_{F,low}$. By induction hypothesis $\mathrm{E}_1[M_0]^{m_j} \in \mathcal{H}_{F,low}$. Then, by Composition of High Expressions (Lemma 4.4.18), $M^{m_j} \in \mathcal{H}_{F,low}$.

$\mathbf{E}[M_0] = (V\ \mathbf{E_1}[M_0])$. Then by App we have $\Sigma;\Gamma \vdash_F^{\check{m}_j} V : s_1, \tau_1 \xrightarrow[F,\check{m}_j]{s_1'} \sigma_1$ and $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1'', \tau_1$ with $s_1''.t \preceq_F s_1'.w$. By Remark 4.4.11 we have $s_1''.t \not\preceq_F low$, and so $s_1'.w \not\preceq_F low$. Therefore, $s_1'.w \not\preceq_F low$, and $s_1''.w \not\preceq_F low$, which means that V is a syntactically (F, low, m_j)-high function and $\mathrm{E}_1[M_0]$ is (F, low, m_j)-high. By induction hypothesis $\mathrm{E}_1[M_0]^{m_j} \in \mathcal{H}_{F,low}$. Then, by Composition of High Expressions (Lemma 4.4.18), $M^{m_j} \in \mathcal{H}_{F,low}$.

$\mathbf{E}[M_0] = (\mathbf{if}\ \mathbf{E_1}[M_0]\ \mathbf{then}\ M_t\ \mathbf{else}\ M_f)$. Then by rule Cond we have that $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \mathsf{bool}$, and $\Sigma;\Gamma \vdash_F^{\check{m}_j} M_t : s_1', \tau_1$ and $\Sigma;\Gamma \vdash_F^{\check{m}_j} M_f : s_1'', \tau_1$ with $s_1.t \preceq_F s_1'.w, s_1'.w$. By Remark 4.4.11 we have $s_1.t \not\preceq_F low$, and so $s_1'.w, s_1'.w \not\preceq_F low$. By High Expressions (Lemma 4.4.9) we have ${M_t}^{m_j}, {M_t}^{m_j} \in \mathcal{H}_{F,low}$. By induction hypothesis $\mathrm{E}_1[M_0]^{m_j} \in \mathcal{H}_{F,low}$. Then, by Composition of High Expressions (Lemma 4.4.18), $M^{m_j} \in \mathcal{H}_{F,low}$.

$\mathbf{E}[M_0] = (\mathbf{E_1}[M_0]; M_1)$. Then by Seq we have that $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \tau_1$ and $\Sigma;\Gamma \vdash_F^{\check{m}_j} M_1 : s_1', \tau_1'$ with $s_1.t \preceq_F s_1'.w$. By Remark 4.4.11 we have $s_1.t \not\preceq_F low$, and so $s_1'.w \not\preceq_F low$. By High Expressions (Lemma 4.4.9) we have ${M_1}^{m_j} \in \mathcal{H}_{F,low}$. By induction hypothesis $\mathrm{E}_1[M_0]^{m_j} \in \mathcal{H}_{F,low}$. Then, by Composition of High Expressions (Lemma 4.4.18), $M^{m_j} \in \mathcal{H}_{F,low}$.

$\mathbf{E}[M_0] = (\mathbf{ref}_{l,\theta}\ \mathbf{E_1}[M_0])$. Then by Ref we have that $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \theta$ with $s_1.t \preceq_F l$. By Remark 4.4.11 we have $s_1.t \not\preceq_F low$, and so $l \not\preceq_F low$. By induction hypothesis $\mathrm{E}_1[M_0]^{m_j} \in \mathcal{H}_{F,low}$. Then, by Composition of High Expressions (Lemma 4.4.18), $M^{m_j} \in \mathcal{H}_{F,low}$.

$\mathbf{E}[M_0] = (?\ \mathbf{E_1}[M_0])$. Easy, by induction hypothesis.

$\mathbf{E}[M_0] = (\mathbf{E_1}[M_0] :=^? M_1)$. Then by Ass we have that $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \theta\ \mathrm{ref}_{\bar{l},\check{n}_{\bar{k}}}$ and $\Sigma;\Gamma \vdash_F^{\check{m}_j} M_1 : s_1', \tau_1$ with $s_1.t \preceq_F s_1'.w$ and $s_1.t \preceq_F l$. By Remark 4.4.11 we have $s_1.t \not\preceq_F \mathit{low}$, and so $\bar{l} \not\preceq_F \mathit{low}$ and $s_1'.w, l \not\preceq_F \mathit{low}$. Hence, by High Expressions (Lemma 4.4.9) we have ${M_1}^{m_j} \in \mathcal{H}_{F,low}$. By induction hypothesis $\mathrm{E}_1[M_0]^{m_j} \in \mathcal{H}_{F,low}$. Then, by Composition of High Expressions (Lemma 4.4.18), $M^{m_j} \in \mathcal{H}_{F,low}$.

$\mathbf{E}[M_0] = (V :=^? \mathbf{E_1}[M_0])$. Then by Ass we have $\Sigma;\Gamma \vdash_F^{\check{m}_j} V : s_1, \theta\ \mathrm{ref}_{\bar{l},\check{n}_{\bar{k}}}$ and $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1', \tau_1$ with $s_1'.t \preceq_F \bar{l}$. By Remark 4.4.11 we have $s_1'.t \not\preceq_F \mathit{low}$, and so $\bar{l} \not\preceq_F \mathit{low}$. By induction hypothesis $\mathrm{E}_1[M_0]^{m_j} \in \mathcal{H}_{F,low}$. Then, by Composition of High Expressions (Lemma 4.4.18), $M^{m_j} \in \mathcal{H}_{F,low}$.

$\mathbf{E}[M_0] = (\mathbf{flow}\ F'\ \mathbf{in}\ \mathbf{E_1}[M_0])$. Then by Flow we have $\Sigma;\Gamma \vdash_{F\cup F'}^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \tau_1$. By induction hypothesis $\mathrm{E}_1[M_0]^{m_j} \in \mathcal{H}_{F\cup F',low}$, which implies $\mathrm{E}_1[M_0]^{m_j} \in \mathcal{H}_{F,low}$. Then, by Composition of High Expressions (Lemma 4.4.18), we conclude that $M^{m_j} \in \mathcal{H}_{F,low}$.

□

We now design a binary relation on expressions that uses $\mathcal{T}_{F,low}^{m_j}$ to ensure that high-terminating expressions are always followed by operationally high ones. The definition of $\mathcal{R}_{G,F,low}^{m_j}$, abbreviated $\mathcal{R}_{F,low}^{m_j}$ when the global flow policy is G, is given in Figure 4.9. 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 $\mathcal{R}_{F,low}^{m_j}$ 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 4.4.22. *If $M_1\ \mathcal{T}_{F,low}^{m_j}\ M_2$, then $M_1\ \mathcal{R}_{F,low}^{m_j}\ M_2$.*

The above remark is used to prove the following lemma.

Lemma 4.4.23. *If for some m_j, F and low we have that $M_1\ \mathcal{R}_{F,low}^{m_j}\ M_2$ and $M_1 \in \mathcal{H}_{F,low}$, then $M_2 \in \mathcal{H}_{F,low}$.*

Rationale. Similarly to Lemma 4.4.19, the proof rests on the fact that if an expression M_1 of the form $(\bar{M}_1\ \bar{N}_1)$, $(\bar{M}_1; \bar{N}_1)$ or $(\bar{M}_1 := \bar{N}_1)$ is operationally high, in spite of $\bar{N}_1$ not being operationally high, then $\bar{M}_1$ is non-resolvable. Clauses 5', 7' and 11' do not require $\bar{N}_1$ to be operationally high. However, by the above observation and by Lemma 4.4.17 this implies that $\bar{M}_2$ is non-resolvable. Therefore, it is sufficient to conclude that $\bar{M}_2$ is operationally high.

Proof. By induction on the definition of $M_1\ \mathcal{R}_{F,low}^{m_j}\ M_2$.

Clause 1'. Direct.

Clause 2'. Direct.

Definition 4.4.21 ($\mathcal{R}^{m_j}_{F,low}$)**.** *We have that* $M_1 \ \mathcal{R}^{m_j}_{F,low} \ M_2$ *if* $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M_1 : s_1, \tau$ *and* $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M_2 : s_2, \tau$ *for some* Σ, Γ, s_1, s_2 *and* τ *and one of the following holds:*

Clause 1'. ${M_1}^{m_j}, {M_2}^{m_j} \in \mathcal{H}_{F,low}$*, or*

Clause 2'. $M_1 = M_2$*, or*

Clause 3'. $M_1 = (\text{if } \bar{M}_1 \text{ then } \bar{N}_t \text{ else } \bar{N}_f)$ *and* $M_2 = (\text{if } \bar{M}_2 \text{ then } \bar{N}_t \text{ else } \bar{N}_f)$ *with* $\bar{M}_1 \ \mathcal{R}^{m_j}_{F,low} \ \bar{M}_2$*, and* ${\bar{N}_t}^{m_j}, {\bar{M}_f}^{m_j} \in \mathcal{H}_{F,low}$*, or*

Clause 4'. $M_1 = (\bar{M}_1 \ \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 \ \bar{N}_2)$ *with* $\bar{M}_1 \ \mathcal{R}^{m_j}_{F,low} \ \bar{M}_2$*, and* $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathcal{H}_{F,low}$*, and* $\bar{M}_1, \bar{M}_2$ *are syntactically* (F, low, m_j)*-high functions, or*

Clause 5'. $M_1 = (\bar{M}_1 \ \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 \ \bar{N}_2)$ *with* $\bar{M}_1 \ \mathcal{T}^{m_j}_{F,low} \ \bar{M}_2$*, and* $\bar{N}_1 \ \mathcal{R}^{m_j}_{F,low} \ \bar{N}_2$*, and* $\bar{M}_1, \bar{M}_2$ *are syntactically* (F, low, m_j)*-high functions, or*

Clause 6'. $M_1 = (\bar{M}_1; \bar{N})$ *and* $M_2 = (\bar{M}_2; \bar{N})$ *with* $\bar{M}_1 \ \mathcal{R}^{m_j}_{F,low} \ \bar{M}_2$*, and* $\bar{N}^{m_j} \in \mathcal{H}_{F,low}$*, or*

Clause 7'. $M_1 = (\bar{M}_1; \bar{N})$ *and* $M_2 = (\bar{M}_2; \bar{N})$ *with* $\bar{M}_1 \ \mathcal{T}^{m_j}_{F,low} \ \bar{M}_2$*, or*

Clause 8'. $M_1 = (\text{ref}_{l,\theta} \ \bar{M}_1)$ *and* $M_2 = (\text{ref}_{l,\theta} \ \bar{M}_2)$ *with* $\bar{M}_1 \ \mathcal{R}^{m_j}_{F,low} \ \bar{M}_2$*, and* $l \not\preceq_F low$*, or*

Clause 9'. $M_1 = (?\ \bar{M}_1)$ *and* $M_2 = (?\ \bar{M}_2)$ *with* $\bar{M}_1 \ \mathcal{R}^{m_j}_{F,low} \ \bar{M}_2$*, or*

Clause 10'. $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ *with* $\bar{M}_1 \ \mathcal{R}^{m_j}_{F,low} \ \bar{M}_2$*, and* $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathcal{H}_{F,low}$*, and* $\bar{M}_1, \bar{M}_2$ *both have type* $\theta \ \text{ref}_{l,\check{n}_k}$ *for some* θ *and* l *such that* $l \not\preceq_F low$*, or*

Clause 11'. $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ *and* $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ *with* $\bar{M}_1 \ \mathcal{T}^{m_j}_{F,low} \ \bar{M}_2$*, and* $\bar{N}_1 \ \mathcal{R}^{m_j}_{F,low} \ \bar{N}_2$*, and* $\bar{M}_1, \bar{M}_2$ *both have type* $\theta \ \text{ref}_{l,\check{n}_k}$ *for some* θ *and* l *such that* $l \not\preceq_F low$*, 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* $\bar{M}_1 \ \mathcal{R}^{m_j}_{F \cup F', low} \ \bar{M}_2$*.*

Figure 4.9: The relation $\mathcal{R}^{m_j}_{F,low}$

Clause 3'. Here we have that $M_1 = (\text{if } \bar{M}_1 \text{ then } \bar{M}_t \text{ else } \bar{M}_f)$ and that $M_2 = (\text{if } \bar{M}_2 \text{ then } \bar{M}_t \text{ else } \bar{M}_f)$ with $\bar{M}_1 \; \mathcal{R}^{m_j}_{F,low} \; \bar{M}_2$ and $\bar{M}_t^{m_j}, \bar{M}_f^{m_j} \in \mathcal{H}_{F,low}$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M}_2 \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $\bar{M}_2 \in \mathcal{H}_{F,low}$.

Clause 4'. Here $M_1 = (\bar{M}_1 \; \bar{N}_1)$ and $M_2 = (\bar{M}_2 \; \bar{N}_2)$ with $\bar{M}_1 \; \mathcal{R}^{m_j}_{F,low} \; \bar{M}_2$, $\bar{M}_1$ and $\bar{M}_2$ are syntactically (F, low, m_j)-high functions, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathcal{H}_{F,low}$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M}_2 \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $\bar{M}_2 \in \mathcal{H}_{F,low}$.

Clause 5'. Here $M_1 = (\bar{M}_1 \; \bar{N}_1)$ and $M_2 = (\bar{M}_2 \; \bar{N}_2)$ with $\bar{M}_1 \; \mathcal{T}^{m_j}_{F,low} \; \bar{M}_2$, $\bar{M}_1$ and $\bar{M}_2$ are syntactically (F, low, m_j)-high functions, and $\bar{N}_1 \; \mathcal{R}^{m_j}_{F,low} \; \bar{N}_2$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by Lemma 4.4.19 also $\bar{M}_2 \in \mathcal{H}_{F,low}$. We distinguish two sub-cases:

- $\boldsymbol{\bar{N}_1 \in \mathcal{H}_{F,low}}$**.** Then, by induction hypothesis, also $\bar{N}_2 \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.
- $\boldsymbol{\bar{N}_1 \notin \mathcal{H}_{F,low}}$**.** Then $\bar{M}_1\dagger$, and by Lemma 4.4.17 also $\bar{M}_2\dagger$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

Clause 6'. Here $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ where $\bar{M}_1 \; \mathcal{R}^{m_j}_{F,low} \; \bar{M}_2$ and $\bar{N}^{m_j} \in \mathcal{H}_{F,low}$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M}_2 \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $\bar{M}_2 \in \mathcal{H}_{F,low}$.

Clause 7'. Here $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1 \; \mathcal{T}^{m_j}_{F,low} \; \bar{M}_2$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by Lemma 4.4.19 also $\bar{M}_2 \in \mathcal{H}_{F,low}$. We distinguish two sub-cases:

- $\boldsymbol{\bar{N} \in \mathcal{H}_{F,low}}$**.** Then, $\bar{M}_2, \bar{N} \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.
- $\boldsymbol{\bar{N} \notin \mathcal{H}_{F,low}}$**.** Then $\bar{M}_1\dagger$, and by Lemma 4.4.17 also $\bar{M}_2\dagger$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

Clause 8'. Here $M_1 = (\text{ref}_{l,\theta} \; \bar{M}_1)$ and $M_2 = (\text{ref}_{l,\theta} \; \bar{M}_2)$ where $\bar{M}_1 \; \mathcal{R}^{m_j}_{F,low} \; \bar{M}_2$, and $l \not\preceq_F low$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M}_2 \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

Clause 9'. Here $M_1 = (?\; \bar{M}_1)$ and $M_2 = (?\; \bar{M}_2)$ where $\bar{M}_1 \; \mathcal{R}^{m_j}_{F,low} \; \bar{M}_2$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M}_2 \in \mathcal{H}_{F,low}$. This implies that $\bar{M}_2 \in \mathcal{H}_{F,low}$.

Clause 10'. Here we have $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ where $\bar{M}_1 \; \mathcal{R}^{m_j}_{F,low} \; \bar{M}_2$, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathcal{H}_{F,low}$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta \; \text{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$. Clearly we have that

$\bar{M}_1 \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M}_2 \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

Clause 11'. Here we have $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ where $\bar{M}_1 \; \mathcal{T}^{m_j}_{F,low} \; \bar{M}_2$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta \; \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$, and $\bar{N}_1 \; \mathcal{R}^{m_j}_{F,low} \; \bar{N}_2$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by Lemma 4.4.19 also $\bar{M}_2 \in \mathcal{H}_{F,low}$. We distinguish two sub-cases:

- $\bar{N} \in \mathcal{H}_{F,low}$. Then, $\bar{M}_2, \bar{N} \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.
- $\bar{N} \notin \mathcal{H}_{F,low}$. Then $\bar{M}_1\dagger$, and by Lemma 4.4.17 also $\bar{M}_2\dagger$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

Clause 12'. Here $M_1 = (\text{flow } F' \text{ in } \bar{M}_1)$ and $M_2 = (\text{flow } F' \text{ in } \bar{M}_2)$ with $\bar{M}_1 \; \mathcal{R}^{m_j}_{F \cup F',low} \; \bar{M}_2$. Clearly we have that $\bar{M}_1 \in \mathcal{H}_{F,low}$, so by induction hypothesis also $\bar{M}_2 \in \mathcal{H}_{F,low}$. Therefore, by Composition of High Expressions (Lemma 4.4.18) we have that $M_2 \in \mathcal{H}_{F,low}$.

□

We have seen in Splitting Computations (Lemma 4.2.1) 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. In Lemma 4.4.24 we saw that the relation $\mathcal{T}^{m_j}_{F,low}$ relates the possible outcomes of expressions that are typable with a low termination effect. Finally, from the following lemma one can conclude that the above relation $\mathcal{R}^{m_j}_{F,low}$ relates the possible outcomes of typable expressions in general.

Lemma 4.4.24. *If there exist* Σ, Γ, s, τ *such that* $\Sigma; \Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}[(?\ a_{l,\theta})] : s, \tau$ *with* $l \not\preceq_{F \cup \lceil \mathrm{E} \rceil} low$*, then for any values* $V_0, V_1 \in \boldsymbol{Val}$ *such that* $\Sigma; \Gamma \vdash V_i : \theta$ *we have* $\mathrm{E}[V_0] \; \mathcal{R}^{m_j}_{F,low} \; \mathrm{E}[V_1]$.

Rationale. 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 4.4.14 any two possible outcomes are in the relation $\mathcal{T}$ (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 Ass); 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 $\mathcal{R}$ 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 '$s.t \preceq_F l'$, where s is the subprogram that performs the high dereference, and l 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 ($s.r \preceq_F l$ in rule Ref); to determine a reference that is being assigned to ($s.r \preceq_F l$ in rule Ass); to determine a value that is being assigned ($s'.r \preceq_F l$ in rule Ass); 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 $s.r \preceq_F s_t.w, s_f.w$ in rule Cond.

Proof. By induction on the structure of E.

$\mathbf{E}[(?\ a_{l,\theta})] = (?\ a_{l,\theta})$. We have $V_0\ \mathcal{R}^{m_j}_{F,low}\ V_1$ by Clause 1'.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{E_1}[(?\ a_{l,\theta})]\ M)$. By rule App we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\bar{\tau} \xrightarrow[F,\Sigma(m_j)]{\bar{s}'} \bar{\sigma}$ and $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F M : \bar{s}'',\bar{\tau}$ with $\bar{s}.r \preceq_F \bar{s}'.w$ and $\bar{s}.t \preceq_F \bar{s}''.w$. By Lemma 4.4.2, we have $l \preceq \bar{s}.r$. Therefore $l \preceq_F \bar{s}'.w$. Since by hypothesis $l \not\preceq_{F\cup\lceil \mathrm{E}_1\rceil}$ *low* (therefore $l \not\preceq_F$ *low*), then $\bar{s}'.w \not\preceq_F$ *low*, that is $\mathrm{E}_1[(?\ a_{l,\theta})]$ is a syntactically (F, low, m_j)-high function. By Lemma 4.4.6, the same holds for $\mathrm{E}_1[V_0]$ and $\mathrm{E}_1[V_1]$. By induction hypothesis we conclude that $\mathrm{E}_1[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$.

- **$\bar{s}.t \not\preceq_F$ *low*.** Then $\bar{s}''.w \not\preceq_F$ *low* (and also $\bar{s}''.w \not\preceq$ *low*) so by High Expressions (Lemma 4.4.9) we have $M^{m_j} \in \mathcal{H}_{F,low}$. Therefore, we conclude $\mathrm{E}[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 4' and Lemma 4.4.6.
- **$\bar{s}.t \preceq_F$ *low*.** Then by Lemma 4.4.14 we have $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. Therefore, since $M\ \mathcal{R}^{m_j}_{F,low}\ M$ by Clause 2', we conclude that $\mathrm{E}[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 5' and Lemma 4.4.6.

$\mathbf{E}[(?\ a_{l,\theta})] = (V\ \mathbf{E_1}[(?\ a_{l,\theta})])$. By rule App we have that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F V : \bar{s},\bar{\tau} \xrightarrow[F,\Sigma(m_j)]{\bar{s}'} \bar{\sigma}$ and $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}'',\bar{\tau}$ with $\bar{s}''.r \preceq_F \bar{s}'.w$. By Lemma 4.4.2, we have $l \preceq \bar{s}''.r$, and so $l \preceq_F \bar{s}'.w$. Since by hypothesis $l \not\preceq_{F\cup\lceil \mathrm{E}_1\rceil}$ *low* (therefore $l \not\preceq_F$ *low*), then $\bar{s}'.w \not\preceq_F$ *low*, that is V is a syntactically (F, low, m_j)-high function. By Clause 1 we have $V\ \mathcal{T}^{m_j}_{F,low}\ V$. By induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. Therefore we conclude that $\mathrm{E}[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 5' and Lemma 4.4.6.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{if}\ \mathbf{E_1}[(?\ a_{l,\theta})]\ \mathbf{then}\ M_t\ \mathbf{else}\ M_f)$. By Cond we have that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \mathsf{bool}$, and $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F M_t : \bar{s}_t,\bar{\tau}$ and $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_F M_f : \bar{s}_f,\bar{\tau}$ with $\bar{s}.r \preceq_F \bar{s}_t.w, \bar{s}_f.w$. By Lemma 4.4.2, we have $l \preceq \bar{s}.r$ and so $l \preceq_F \bar{s}_t.w, \bar{s}_f.w$. Since by hypothesis $l \not\preceq_{F\cup\lceil \mathrm{E}_1\rceil}$ *low*

(therefore $l \not\preceq_F low$), then $\bar{s}_t.w \not\preceq_F low$ and $\bar{s}_f.w \not\preceq_F low$. This implies that ${M_t}^{m_j}, {M_f}^{m_j} \in \mathcal{H}_{F,low}$. By induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. Therefore we conclude that $\mathrm{E}[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 3' and Lemma 4.4.6.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{E_1}[(?\ a_{l,\theta})]; M)$. By SEQ we have $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\bar{\tau}$ and $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} M : \bar{s}',\bar{\tau}'$ with $\bar{s}.t \preceq_F \bar{s}'.w$.

- **$\bar{s}.t \not\preceq_F low$.** Then $\bar{s}'.w \not\preceq_F low$ so by High Expressions (Lemma 4.4.9) we have $M^{m_j} \in \mathcal{H}_{F,low}$. By induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. We then conclude that $\mathrm{E}[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 6' and Lemma 4.4.6.
- **$\bar{s}.t \preceq_F low$.** Then by Lemma 4.4.14 we have $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. Therefore, we conclude using Clause 7' and Lemma 4.4.6.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{ref}_{\bar{l},\bar{\theta}}\ \mathbf{E_1}[(?\ a_{l,\theta})])$. By REF we have $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\bar{\tau}$ with $\bar{s}.r = s.r \preceq_F \bar{l}$ and $\bar{s}.t = s.t$. Therefore, since $l \not\preceq_{F\cup \mathrm{E}} low$ implies $l \not\preceq_{F\cup \mathrm{E}_1} low$, then by induction hypothesis we have $\mathrm{E}_1[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. By Lemma 4.4.2 we have $l \preceq s.r$, so $s.r \not\preceq_F low$. Therefore, $\bar{l} \not\preceq_F low$, and we conclude by Lemma 4.4.6 and Clause 8'.

$\mathbf{E}[(?\ a_{l,\theta})] = (?\ \mathbf{E_1}[(?\ a_{l,\theta})])$. By rule DER we have $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\bar{\tau}$. By induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. We conclude by Lemma 4.4.6 and Clause 9'.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{E_1}[(?\ a_{l,\theta})] :=^? M)$. By rule ASS we have that $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[a_{l,\theta}] : \bar{s},\bar{\theta}\ \mathrm{ref}_{\bar{l},\check{\bar{n}}_k}$ with $\bar{s}.r \preceq_F \bar{l}$ and $\bar{s}.t \preceq_F \bar{s}'.w$. By Lemma 4.4.2 we have $l \preceq s.r$, so $s.r \not\preceq_F low$ and so $\bar{l} \not\preceq_F low$.

- **$\bar{s}.t \not\preceq_F low$.** Then $\bar{s}'.w \not\preceq_F low$ so by High Expressions (Lemma 4.4.9) we have $M^{m_j} \in \mathcal{H}_{F,low}$. By induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. We then conclude that $\mathrm{E}[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 10' and Lemma 4.4.6.
- **$\bar{s}.t \preceq_F low$.** Then by Lemma 4.4.14 we have $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. Therefore, we conclude using Lemma 4.4.6, Clause 11' and Clause 2' (regarding M).

$\mathbf{E}[(?\ a_{l,\theta})] = (V :=^? \mathbf{E_1}[(?\ a_{l,\theta})])$. By rule ASS we have that $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} V : \bar{s},\bar{\theta}\ \mathrm{ref}_{\bar{l},\check{\bar{n}}_k}$, $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[a_{l,\theta}] : \bar{s}',\theta$ with $\bar{s}'.r \preceq_F \bar{l}$. By Lemma 4.4.2 we have $l \preceq \bar{s}'.r$, so $l \preceq_F \bar{l}$. Then, we must have $\bar{l} \not\preceq_F low$, since otherwise $l \preceq_{F\cup \mathrm{E}} low$. By Clause 1 we have that $V\ \mathcal{T}^{m_j}_{F,low}\ V$, and by induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. We then conclude by Lemma 4.4.6 and Clause 11'.

$\mathbf{E}[(?\ a_{l,\theta})] = (\mathbf{flow}\ F'\ \mathbf{in}\ \mathbf{E_1}[(?\ a_{l,\theta})])$. By rule FLOW we have $\Sigma;\Gamma \vdash_{F\cup F'}^{\Sigma(m_j)} V : s,\tau$. By induction hypothesis $\mathrm{E}_1[V_0]\ \mathcal{T}^{m_j}_{F\cup F',low}\ \mathrm{E}_1[V_1]$, so we conclude by Lemma 4.4.6 and Clause 12'.

□

We now state a crucial result of the paper: the relation $\mathcal{R}^{m_j}_{F,low}$ is a sort of "strong bisimulation".

Proposition 4.4.25 (Strong Bisimulation for Typable Low Threads)**.** *If* $M_1 \ \mathcal{R}^{m_j}_{F,low} \ M_2$ *and* $M_1 \notin \mathcal{H}_{F,low}$ *and* $\langle {M_1}^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle {M_1'}^{m_j}, T_1', S_1'\rangle$, *with* $\langle T_1, S_1\rangle =^{F\cup F', low} \langle T_2, S_2\rangle$ *such that* n *is fresh for* T_2 *if* $n \in \mathrm{dom}(T_1' - T_1)$ *and* a *is fresh for* S_2 *if* $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$, *then there exist* T_2', M_2' *and* S_2' *such that* $\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{n_k}} \langle {M_2'}^{m_j}, T_2', S_2'\rangle$ *with* $M_1' \ \mathcal{R}^{m_j}_{F,low} \ M_2'$ *and* $\langle T_1', S_1'\rangle =^{F,low} \langle T_2', S_2'\rangle$.

Rationale. Assuming that M_1 (and M_2) are not operationally high allows us to conclude in many cases that a certain subexpression can compute, using Composition of High Expressions (Lemma 4.4.18). It applies in particular to potentially suspensive expressions.
If $M_1^{m_j}$ and $M_2^{m_j}$ are equal (related by $\mathcal{T}$ using Clause 2), then we can reject the case where m_j is accessing a high remote reference, since by Potentially Suspensive Transitions (Lemma 4.4.20) we would have $M_1^{m_j}$ operationally high. We can then proceed as in Strong Bisimulation for Low-Terminating Threads (4.4.15).

Proof. By induction on the definition of $\mathcal{R}^{m_j}_{F,low}$. We use Subject Reduction (Theorem 4.4.7) (Theorem 4.4.7) to guarantee typability (with the same type) for m_j, *low* and F, which is a requirement for being in the $\mathcal{R}^{m_j}_{F,low}$ relation. We also use the Strong Bisimulation for Low Terminating Threads Lemma (Lemma 4.4.15)

Clause 1'. This case is excluded by assumption.

Clause 2'. Here $M_1 = M_2$. If $M = \mathrm{E}[(n_k.u_{l,\theta} :=^? V)]$ or $M = \mathrm{E}[(?\ n_k.u_{l,\theta})]$ with $j \curlyvee k \not\preceq_F$ *low* and $n \neq m$, then by Potentially Suspensive Transitions (Lemma 4.4.20) we have that $M^{m_j} \in \mathcal{H}_{F,low}$, which is rejected by assumption. Otherwise, the proof is analogous to the corresponding case in Strong Bisimulation for Low-Typable Threads (Lemma 4.4.15): By Guaranteed Transitions (Lemma 4.4.10) there exist T_2', M_2' and S_2' such that $\langle M_2^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{n_k}} \langle {M_2'}^{m_j}, T_2', S_2'\rangle$ with $\langle T_1', S_1'\rangle =^{F\cup F', low} \langle T_2', S_2'\rangle$.

- $\boldsymbol{M_2' = M_1'}$**.** Then we have $M_1' \ \mathcal{R}^{m_j}_{F,low} \ M_2'$, by Clause 2' and Subject Reduction (Theorem 4.4.7).
- $\boldsymbol{M_2' \neq M_1'}$**.** Then by Splitting Computations (Lemma 4.2.1) we have that $(N^{n_k} = ())$ and there exists E and $a_{l,\theta}$ such that $F' = \lceil \mathrm{E} \rceil$, $M_1' = \mathrm{E}[S_1(a_{l,\theta})]$, $M_2' = \mathrm{E}[S_2(a_{l,\theta})]$, $\langle T_1', S_1'\rangle = \langle T_1, S_1\rangle$ and $\langle T_2', S_2'\rangle = \langle T_2, S_2\rangle$. Since $S_1(a_{l,\theta}) \neq S_2(a_{l,\theta})$, we have $l \not\preceq_{F\cup F'}$ *low*. Therefore, $M_1' \ \mathcal{R}^{m_j}_{F,low} \ M_2'$, by Lemma 4.4.24 above.

Clause 3'. Here we have that $M_1 = (\text{if } \bar{M}_1 \text{ then } \bar{M}_t \text{ else } \bar{M}_f)$ and that $M_2 = (\text{if } \bar{M}_2 \text{ then } \bar{M}_t \text{ else } \bar{M}_f)$ with $\bar{M}_1 \ \mathcal{R}^{m_j}_{F,low} \ \bar{M}_2$ and $\bar{M}_t^{m_j}, \bar{M}_f^{m_j} \in \mathcal{H}_{F,low}$. We can assume that $\bar{M}_1^{m_j} \notin \mathcal{H}_{F,low}$, since otherwise $M_1^{m_j} \in \mathcal{H}_{F,low}$ by

Composition of High Expressions (Lemma 4.4.18). Therefore, $M_1' = (\text{if } \bar{M}_1' \text{ then } \bar{M}_t \text{ else } \bar{M}_f)$ with $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use the induction hypothesis, Clause 3' and Subject Reduction (Theorem 4.4.7) to conclude.

Clause 4'. Here $M_1 = (\bar{M}_1\ \bar{N}_1)$ and $M_2 = (\bar{M}_2\ \bar{N}_2)$ with $\bar{M}_1\ \mathcal{R}^{m_j}_{F,low}\ \bar{M}_2$, $\bar{M}_1$ and $\bar{M}_2$ are syntactically (F, low, m_j)-high functions, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathcal{H}_{F,low}$. We can assume that $\bar{M}_1$ can compute, since otherwise $M_1^{m_j} \in \mathcal{H}_{F,low}$ by Composition of High Expressions (Lemma 4.4.18). Therefore, $M_1' = (\bar{M}_1'\ \bar{N}_1)$ with $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use the induction hypothesis, Clause 4' and Subject Reduction (Theorem 4.4.7) to conclude.

Clause 5'. Here $M_1 = (\bar{M}_1\ \bar{N}_1)$ and $M_2 = (\bar{M}_2\ \bar{N}_2)$ with $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$, $\bar{M}_1$ and $\bar{M}_2$ are syntactically (F, low, m_j)-high functions, and $\bar{N}_1\ \mathcal{R}^{m_j}_{F,low}\ \bar{N}_2$. We distinguish two sub-cases:

- **$\bar{M}_1$ can compute.** In this case exists $\bar{M}_1'$ such that $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use Lemma 4.4.15, Subject Reduction (Theorem 4.4.7) and Clause 5' to conclude.
- **$\bar{M}_1$ is a value.** Then by Remark 4.4.13, $\bar{M}_2 \in \boldsymbol{Val}$. We can assume that $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \notin \mathcal{H}_{F,low}$, since otherwise $M_1^{m_j} \in \mathcal{H}_{F,low}$ by Composition of High Expressions (Lemma 4.4.18). Then, $\bar{N}_1$ can compute, and so there exist $\bar{N}_1'$ such that $\langle \bar{N}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{N}_1'^{m_j}, T_1', S_1'\rangle$ with $M_1' = (\bar{M}_1\ \bar{N}_1')$. We use the induction hypothesis, Clause 5' and Subject Reduction (Theorem 4.4.7) to conclude.

Clause 6'. Here $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ where $\bar{M}_1\ \mathcal{R}^{m_j}_{F,low}\ \bar{M}_2$ and $\bar{N}^{m_j} \in \mathcal{H}_{F,low}$. We can assume that $\bar{M}_1^{m_j} \notin \mathcal{H}_{F,low}$, since otherwise $M_1^{m_j} \in \mathcal{H}_{F,low}$ by Composition of High Expressions (Lemma 4.4.18). Therefore, we have $M_1' = (\bar{M}_1'; \bar{N})$ with $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use the induction hypothesis, Clause 6' and Subject Reduction (Theorem 4.4.7) to conclude.

Clause 7'. Here $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$. We distinguish two sub-cases:

- **$\bar{M}_1$ can compute.** In this case exists $\bar{M}_1'$ such that $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use Lemma 4.4.15, Subject Reduction (Theorem 4.4.7) and Clause 7' to conclude.
- **$\bar{M}_1$ is a value.** Then $M_1' = \bar{N}$, $F = \emptyset$, $N^{n_k} = ()$ and $\langle T_1', S_1'\rangle = \langle T_1, S_1\rangle$. By Remark 4.4.13, $\bar{M}_2 \in \boldsymbol{Val}$. Then, we have $\langle M_2^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{N}^{m_j}, T_1', S_1'\rangle$. We conclude using Lemma 4.4.15 and Clause 2'.

Clause 8'. Here $M_1 = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ and $M_2 = (\mathrm{ref}_{l,\theta}\ \bar{M}_2)$ where $\bar{M}_1\ \mathcal{R}^{m_j}_{F,low}\ \bar{M}_2$, and $l \not\preceq_F low$. We can assume that $\bar{M}_1^{m_j} \notin \mathcal{H}_{F,low}$, since otherwise

$M_1^{m_j} \in \mathcal{H}_{F,low}$ by Composition of High Expressions (Lemma 4.4.18). Then, $\bar{M}_1$ can compute, and $M_1' = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ with $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use the induction hypothesis, Subject Reduction (Theorem 4.4.7) and Clause 8' to conclude.

Clause 9'. Here $M_1 = (?\ \bar{M}_1)$ and $M_2 = (?\ \bar{M}_2)$ where $\bar{M}_1\ \mathcal{R}^{m_j}_{F,low}\ \bar{M}_2$. We know that $\bar{M}_1$ can compute, since otherwise ${M_1}^{m_j} \in \mathcal{H}_{F,low}$. Then, we have $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use the induction hypothesis, Subject Reduction (Theorem 4.4.7) and Clause 9' to conclude.

Clause 10'. Here we have $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ where $\bar{M}_1\ \mathcal{R}^{m_j}_{F,low}\ \bar{M}_2$, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathcal{H}_{F,low}$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$. We can assume that $\bar{M}_1$ can compute, since otherwise $M_1^{m_j} \in \mathcal{H}_{F,low}$ by Composition of High Expressions (Lemma 4.4.18). Therefore, $M_1' = (\bar{M}_1' :=^? \bar{N}_1)$ with $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use the induction hypothesis, Clause 10' and Subject Reduction (Theorem 4.4.7) to conclude.

Clause 11'. Here we have $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ where $\bar{M}_1\ \mathcal{T}^{m_j}_{F,low}\ \bar{M}_2$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$, and $\bar{N}_1\ \mathcal{R}^{m_j}_{F,low}\ \bar{N}_2$. We can assume that M_1 cannot be a redex, with $\bar{M}_1, \bar{N}_1 \in \boldsymbol{Val}$, since otherwise $M_1^{m_j} \in \mathcal{H}_{F,low}$ by Composition of High Expressions (Lemma 4.4.18). There are two cases to consider:

- **$\bar{M}_1$ can compute.** Then we have $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$. We use Lemma 4.4.15, Clause 11' and Subject Reduction (Theorem 4.4.7) to conclude.
- **$\bar{M}_1$ is a value but $\bar{N}_1$ can compute.** Then by Remark 4.4.13, $\bar{M}_2 \in \boldsymbol{Val}$. Then we have $\langle \bar{N}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{N}_1'^{m_j}, T_1', S_1'\rangle$. We conclude using induction hypothesis, Clause 11' and Subject Reduction (Theorem 4.4.7).

Clause 12'. Here $M_1 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_1)$ and $M_2 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_2)$ with $\bar{M}_1\ \mathcal{R}^{m_j}_{F\cup F',low}\ \bar{M}_2$. We can assume that $\bar{M}_1^{m_j} \notin \mathcal{H}_{F\cup F',low}$, since otherwise $\bar{M}_1^{m_j} \notin \mathcal{H}_{F,low}$ and by Composition of High Expressions (Lemma 4.4.18) $M_1^{m_j} \in \mathcal{H}_{F,low}$. Therefore $\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F'']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$ with $F' = \bar{F} \cup F''$. By induction hypothesis, we have that $\langle \bar{M}_2^{m_j}, T_2, S_2\rangle \xrightarrow[F'']{N^{n_k}} \langle \bar{M}_2'^{m_j}, T_2', S_2'\rangle$, and that $M_1'\ \mathcal{R}^{m_j}_{F\cup\bar{F},low}\ M_2'$ and also $\langle T_1', S_1'\rangle =^{F\cup\bar{F},low} \langle T_2', S_2'\rangle$. Notice that $\langle T_1', S_1'\rangle =^{F,low} \langle T_2', S_2'\rangle$. We use Subject Reduction (Theorem 4.4.7) and Clause 12' to conclude.

□

Behavior of Sets of Typable Threads

To conclude the proof of the Soundness Theorem, it remains to exhibit an appropriate bisimulation on pools of threads.

The definition of $\mathcal{R}^{\star}_{G,F,low}$, abbreviated $\mathcal{R}^{m_j}_{F,low}$ when the global flow policy is G, is given in Figure 4.9. The flow policy F is assumed to contain G.

Definition 4.4.26 ($\mathcal{R}^{\star}_{G,low}$)**.** *The relation $\mathcal{R}^{\star}_{low}$ is inductively defined as follows:*

$$a)\ \frac{M^{m_j} \in \mathcal{H}_{G,low}}{\{M^{m_j}\}\ \mathcal{R}^{\star}_{G,low}\ \emptyset} \quad b)\ \frac{M^{m_j} \in \mathcal{H}_{G,low}}{\emptyset\ \mathcal{R}^{\star}_{G,low}\ \{M^{m_j}\}} \quad c)\ \frac{M_1\ \mathcal{R}^{m_j}_{G,low}\ M_2}{\{{M_1}^{m_j}\}\ \mathcal{R}^{\star}_{G,low}\ \{{M_2}^{m_j}\}}$$

$$d)\ \frac{P_1\ \mathcal{R}^{\star}_{G,low}\ P_2 \quad Q_1\ \mathcal{R}^{\star}_{G,low}\ Q_2}{P_1 \cup Q_1\ \mathcal{R}^{\star}_{G,low}\ P_2 \cup Q_2}$$

Proposition 4.4.27. *The relation $\mathcal{R}^{\star}_{G,low}$ is a (G, low)-bisimulation.*

Rationale. Operationally high threads can be added to any pool of threads without affecting its capability of being bisimilar to another pool of threads. This results from the fact that threads in the set $\mathcal{H}$ can only generate threads that are in $\mathcal{H}$, and none of them can perform changes to the low memory. Therefore, any step that is performed by an operationally high thread can be simulated by any pool of threads by doing nothing.
For each of the pairs of threads that are related by $\mathcal{R}^{\star}$, we use Strong Bisimulation for Typable Low Threads (Proposition 4.4.25), and Clause 2' to prove that the expressions that are related by $\mathcal{R}^{m_j}_{G,low}$ can simulate each other's steps, and that any threads that they eventually create are related by $\mathcal{R}^{\star}$.

Proof. First, it is easy to see, by induction on the definition of $\mathcal{R}^{\star}_{G,low}$, that this relation is symmetric. Now we show, by induction on the definition of $\mathcal{R}^{\star}_{G,low}$, that if $P_1\ \mathcal{R}^{\star}_{G,low}\ P_2$ and $\langle P_1, T_1, S_1\rangle \xrightarrow[F]{} \langle P_1', T_1', S_1'\rangle$, n is fresh for T_2 if $n \in \mathrm{dom}(T_1' - T_1)$ and a is fresh for S_2 if $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$, and if $\langle T_1, S_1\rangle =^{F\cup G,low} \langle T_2, S_2\rangle$, then there exist T_2', P_2' and S_2' such that $\langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2', S_2'\rangle$ and $P_1'\ \mathcal{R}^{\star}_{G,low}\ P_2'$ and $\langle T_1', S_1'\rangle =^{F\cup G,low} \langle T_2', S_2'\rangle$.

Rule *a*). Then $P_1 = \{M^{m_j}\}$, $P_2 = \emptyset$, and $M^{m_j} \in \mathcal{H}_{G,low}$. In this case $\langle M^{m_j}, T_1, S_1\rangle \xrightarrow[F]{\emptyset} \langle {M'}^{m_j}, T_1', S_1'\rangle$, with $P_1' = \{{M'}^{m_j}, N^{n_k}\}$, where we have $M_1'^{m_j}, N^{n_k} \in \mathcal{H}_{G,low}$ and $\langle T_1', S_1'\rangle =^{G,low} \langle T_1, S_1\rangle$. We have that $\langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2, T_2, S_2\rangle$ and by transitivity $\langle T_1', S_1'\rangle =^{G,low} \langle T_2, S_2\rangle$. By Rule a) we have $\{M_1'^{m_j}\}\ \mathcal{R}^{\star}_{G,low}\ \emptyset$ and $\{N^{n_k}\}\ \mathcal{R}^{\star}_{G,low}\ \emptyset$. Therefore, by Rule d), we have $P_1'\ \mathcal{R}^{\star}_{G,low}\ \emptyset$.

Rule *c*). Then $P_1 = \{{M_1}^{m_j}\}$ and $P_2 = \{{M_2}^{m_j}\}$, and we have $M_1\ \mathcal{R}^{m_j}_{G,low}\ M_2$. By the case for Rule a), we have that $P_1'\ \mathcal{R}^{\star}_{G,low}\ \emptyset$ and $\langle T_1', S_1'\rangle =^{F\cup G,low} \langle T_2', S_2'\rangle$. Since $M_1^{m_j} \in \mathcal{H}_{G,low}$, then by Lemma 4.4.23 also $M_2^{m_j} \in \mathcal{H}_{G,low}$, so by Rule b) $\emptyset\ \mathcal{R}^{\star}_{G,low}\ P_2$. Then, by Rule d), we have $P_1'\ \mathcal{R}^{\star}_{G,low}\ P_2$. If $M_1^{m_j} \notin \mathcal{H}_{G,low}$, there are two cases to be considered:

$P_1' = \{M_1'^{m_j}\}$. Then $\langle {M_1}^{m_j}, T_1, S_1\rangle \xrightarrow[F]{\emptyset} \langle {M_1'}^{m_j}, T_1', S_1'\rangle$ and so by Strong Bisimulation for Typable Low Threads (Proposition 4.4.25) there exist T_2', M_2' and S_2' such that $\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{\emptyset} \langle {M_2'}^{m_j}, T_2', S_2'\rangle$ with

$M_1' \ \mathcal{R}^{m_j}_{G,low} \ M_2'$ and $\langle T_1', S_1'\rangle =^{G,low} \langle T_2', S_2'\rangle$. Then, by Rule c), we have $\{{M_1'}^{m_j}\} \ \mathcal{R}^{\star}_{G,low} \ \{{M_2}^{m_j}\}$.

$P_1' = \{M_1'^{m_j}, N^{n_k}\}$. Then we have $\langle {M_1}^{m_j}, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle {M_1'}^{m_j}, T_1', S_1'\rangle$ and again by Strong Bisimulation for Typable Low Threads (Proposition 4.4.25) there exist T_2', M_2' and S_2' such that $\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{n_k}} \langle {M_2'}^{m_j}, T_2', S_2'\rangle$ with $M_1' \ \mathcal{R}^{m_j}_{G,low} \ M_2'$ and $\langle T_1', S_1'\rangle =^{G,low} \langle T_2', S_2'\rangle$. Then, by Rule c) we have $\{{M_1}^{m_j}\} \ \mathcal{R}^{\star}_{G,low} \ \{{M_2}^{m_j}\}$. By Subject Reduction (Theorem 4.4.7), by Lemma 4.4.4, and by Clause 2' we have $N \ \mathcal{R}^{n_k}_{G,low} \ N$, and so by Rule c) we have $\{N^{n_k}\} \ \mathcal{R}^{\star}_{G,low} \ \{N^{n_k}\}$. Therefore, by Rule d), we have $\{{M_1}^{m_j}, N^{n_k}\} \ \mathcal{R}^{\star}_{G,low} \ \{{M_2}^{m_j}, N^{n_k}\}$.

Rule *d*). Then $P_1 = \bar{P}_1 \cup \bar{Q}_1$ and $P_2 = \bar{P}_2 \cup \bar{Q}_2$, with $\bar{P}_1 \ \mathcal{R}^{\star}_{G,low} \ \bar{P}_2$ and $\bar{Q}_1 \ \mathcal{R}^{\star}_{G,low} \ \bar{Q}_2$. Suppose that $\langle \bar{P}_1, T_1, S_1\rangle \xrightarrow[F]{} \langle \bar{P}_1', T_1', S_1'\rangle$ – the case where $\bar{Q}_1$ reduces is analogous. By induction hypothesis, there exist T_2', $\bar{P}_2'$ and S_2' such that $\langle \bar{P}_2, T_2, S_2\rangle \twoheadrightarrow \langle \bar{P}_2', T_2', S_2'\rangle$ with $\bar{P}_1' \ \mathcal{R}^{\star}_{G,low} \ \bar{P}_2'$ and $\langle T_1', S_1'\rangle =^{G,low} \langle T_2', S_2'\rangle$. Then, $\langle \bar{P}_2 \cup \bar{Q}_2, T_2, S_2\rangle \twoheadrightarrow \langle \bar{P}_2' \cup \bar{Q}_2, T_2', S_2'\rangle$, and by Rule d) we have $\bar{P}_1' \cup \bar{Q}_1 \ \mathcal{R}^{\star}_{G,low} \ \bar{P}_2' \cup \bar{Q}_2$.

□

We now state the main result of this chapter:

Theorem 4.4.28 (Soundness for Non-disclosure for Networks.)**.** *Consider a pool of threads P and a global flow policy G. If for all $M^{m_j} \in P$ there exist Σ, Γ, s and τ such that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{G,G} M : s,\tau$, then P satisfies the Non-disclosure for Networks policy with respect to G.*

Proof. By Clause 2' of Definition 4.4.21, for all choices of security levels *low*, we have that $M \ \mathcal{R}^{m_j}_{G,Glow} \ M$. By Rule c) of Definition 4.4.26 we then have $\{M^{m_j}\} \ \mathcal{R}^{\star}_{G,low} \ \{M^{m_j}\}$. Since this is true for all $M^{m_j} \in P$, by Rule d) we have that $P \ \mathcal{R}^{\star}_{G,low} \ P$. By Proposition 4.4.27 we conclude that $P \approx_{G,low} P$. □

The above result is compositional, in the sense that it is enough to verify the typability of each thread separately in order to ensure non-disclosure for the whole network. The global flow policy G can be taken as the "intersection" of the flow policies of all the threads in the network. As was observed earlier this operation seems too costly and complex to be used in a general case. The result can be conveniently approximated by the empty flow relation, which gives the minimum flow security pre-lattice that all threads must satisfy.

4.5 Related Work

To the best of our knowledge, this thesis is the first to study insecure information flows that are introduced by mobility in the context of a distributed language with states. Moreover, it seems to be the first to consider the usage of declassification in a distributed scenario.

A first step towards the study of confidentiality for distributed systems is to study a language with concurrency. As we have mentioned earlier in Section 2.5, Smith and Volpano [Smith & Volpano, 1998] considered non-interference for an

imperative multi-threaded language. They recognized termination leaks as an issue that is specific to concurrent settings, but that is not problematic in sequential settings. This line of study was pursued by considering increasingly expressive languages and refined type systems [Smith, 2001; Boudol & Castellani, 2002; Honda & Yoshida, 2002; Almeida Matos & Boudol, 2005; Boudol, 2005b]. In the setting of synchronous concurrent systems, new kinds of termination leaks – the *suspension leaks* – are to be handled. A few representative studies include [Sabelfeld, 2001; Almeida Matos *et al.*, 2004]. The discussion on related work will proceed by focusing on type-based approaches for enforcing information flow control policies in settings with distribution and mobility.

4.5.1 Distribution

Already in a distributed setting, but where interaction between domains is restricted to the exchange of values (no code mobility), Mantel and Sabelfeld [Mantel & Sabelfeld, 2004; Sabelfeld & Mantel, 2002] have provided a type system for preserving confidentiality for different kinds of channels established over a publicly observable medium.

Sharing our underlying aim of studying the distribution of code under decentralized security policies, Zdancewic *et al.* [Zdancewic *et al.*, 2002] have however set the problem in a very different manner. They have considered a distributed system of potentially corrupted hosts and of principals that have different levels of trust on these hosts. They then proposed a way of partitioning the program and distributing the resulting parts over hosts that are trusted by the concerned principals.

4.5.2 Mobility

Progressing rather independently we find a field of work on mobile calculi that are purely functional concurrent languages. In [Kırlı, 2000], mobility of functions as values is studied for a deterministic language with only two sites. To mention a few representative works on process calculi, we have Honda *et al.*'s paper on for π-calculus [Honda *et al.*, 2000], and Hennessy and Riely's study for the security π-calculus [Hennessy & Riely, 2002].

Castagna, Bugliesi and Craffa seem to have been the first to approach the study of non-interference for a language with distribution and mobility [Crafa *et al.*, 2002]. This work was done for Boxed Ambients [Bugliesi *et al.*, 2001], a purely functional process calculus derived from Mobile Ambients [Cardelli & Gordon, 2000] that the authors had previously used as a framework for distributed resource access security [Bugliesi *et al.*, 2001]. Non-interference is stated by means of a contextual equivalence and a sound type system is presented. However, a unique lattice representing the flow policy was considered, and no declassification mechanisms are contemplated.

Distribution in Boxed Ambients (abbreviated BA) is hierarchical, where mobility consists of having *ambients* (n) enter or exit the boundaries of neighboring or parent ambients, (respectively by means of the 'in n' and 'out n' primitives). Communication can occur locally via an unnamed channel ('$\langle M \rangle$' for output of M, '$(x)P$' for an input followed by P), or across boundaries, between parent

and child, via a channel with the child's name ('$(x)^n P$' and '$\langle M \rangle^n P$' for communication with the child n, '$(x)^\uparrow P$' and '$\langle M \rangle^\uparrow P$' for communication with the parent). The execution of both the communication and migration instructions depend on the presence of ambients at certain (neighboring) positions, and are otherwise suspended (in the same sense that our dereference and assignment operations suspend in the absence of the thread they belong to).

Since ambient names correspond simultaneously to places of computation, to subjects of migration, and to resources for passing values, and because it is purely functional, it is hard to establish a correspondence between BA and the language in this thesis. However, some analogies can be drawn at this point. Roughly speaking, security levels are associated to ambient names (we write n_l for an ambient n that has security level l), as they are here to references and threads. Similarly to this thesis, the knowledge of the position of an ambient of level l is considered as l-level information. Message passing between parent and child involves a synchronization that respects the position of those two "domains", as it happens here for the accesses to foreign references. Migration is also identified as a way of revealing the position of "high-ambients" to lower levels, though the dangerous usages of migration are rejected rather differently. To facilitate the comparison, we will now exhibit a couple of programs (from [Crafa *et al.*, 2002]) that are insecure in BA, and explain them in light of the concepts that were used in this thesis. Comparisons on the security property and type system are left for future work.

In the following insecure program, the high ambient n is located inside the low ambient m. Ambient m should perform an input on a channel that belongs to n, and then output some expression M over its outward channel. Ambient n should output expression N over its outward channel. At the top level there is an expression that can read over a channel that belongs to m and then perform P.

$$m_L[(x)^{n_H}.\langle M \rangle^\uparrow \parallel n_H[\langle N \rangle^\uparrow]] \;\parallel\; (x)^{m_L}.P \tag{4.23}$$

Notice that P can only be performed after a communication has been established over m_L's channel. In turn, this can only happen after a communication has been established in m over n_H's channel. Therefore, the execution of P depends on the transmission of information about the location of the high ambient n, over a low channel.

In another example, migration plays an important role. Two ambients m and n are placed at the top level, besides an expression that is willing to receive a value through a channel that belongs to m, before it can execute P. Ambient m contains instructions to first enter into n and then to exit it, after which expression M should be outputted over its outward channel.

$$(x)^{m_L}.P \;\parallel\; m_L[\text{in } n_H.\text{out } n_H.\langle M \rangle^\uparrow] \;\parallel\; n_H[\,] \tag{4.24}$$

Also in this example, P can only be performed after a communication has been established over (low) m's channel. This can only happen if the high level ambient n is located at the same level as the low ambient m. Therefore, the execution of P depends on the transmission of information about the location of the high ambient n, over a low channel.

Chapter 5

Conclusion

In this final chapter we summarize the main technical contributions of this thesis, and give some perspective on future work. We end with some concluding remarks on the main ideas introduced in this thesis.

5.1 Main Contributions And Future Work

Security Policies We have addressed the issue of what is a secure program from the point of view of confidentiality in information flow. We studied two main security policies: the classical non-interference property, that determines the absence of information flows that are insecure according to a static and global ordering of security levels and a new non-disclosure property, that determines the absence of information flows that are insecure according to a dynamically chosen ordering of security levels that is valid at that point.

Non-interference and non-disclosure policies were defined in terms of bisimulations, naturally based on small step transitions. This provides the necessary refinement to, on one hand, scrutinize the changes in memory that occur at each computation step (necessary in concurrent settings), and on the other, to pinpoint the valid flow policy (necessary to restrict the scope of the flow policy declarations) by decorating the small step semantics. We show that the orderings of security levels can be expressed in terms of simple flow policies, like relations on principals.

We believe that non-disclosure is a natural generalization of classical non-interference, and that the idea of using bisimulations on labeled small step semantics to state a security property that reflects the *local* nature of declassification could perhaps be used in other settings. For instance, following Biba's remark that integrity is in a sense dual to confidentiality (see [Li *et al.*, 2003; Myers & Liskov, 1997]), we could design a similar framework for the integrity aspect of security, possibly including downgrading facilities like the "endorse" constructs of [Li *et al.*, 2003; Myers *et al.*, 2004].

Computation Models We have directed our study towards two main concurrency paradigms. One, in a local setting, where threads can be created dynamically and execute in parallel in the same computational medium. Another, in a distributed setting with thread migration, where threads execute in

different domains, and the relative location of threads and resources determines the circumstances in which they can execute. In the latter, we found that new forms of security leaks – the migration leaks – could be encoded. Some resemblance was found with the leaks of information which can be caused by the position of threads in an ambient-like network [Crafa *et al.*, 2002]. Indeed, in that work, the visibility of threads in a network is also considered to be subject to confidentiality requirements. This seems to indicate that our results are not confined to our particular network model.

We purposely chose a simple model of mobility, sufficient to expose the principles behind migration leaks[1]. Nevertheless, one can expect more complex models of global computing to have interesting impacts on the study of information flow control. For instance, having a more general form of migration that can be induced by a thread upon another (objective migration) is likely to bring out new ways of expressing migration leaks. On the other hand, introducing membrane computation [Boudol, 2005a] as a prerequisite for a thread to enter a domain could provide for new ways of controlling information leaks.

One prominent research direction in models for global computing addresses the failure-prone nature of networks. As was pointed out in [Boudol, 2004], the principles of reactive systems seem particularly suitable for providing forms of reaction to failures. The ULM language that was presented there shows how this can be done. It could be interesting to see the impact of similarly integrating the reactive principles in the model that we used in this thesis.

Language Features The languages upon which we performed our study are simple but expressive. Our starting point was an imperative higher-order lambda-calculus with thread and reference creation. The core language was then enriched with a flow declaration construct that allows a dynamic customization of the security ordering. A location-aware version of the core language, with a migration instruction that changes the position of a thread and its references, was considered last.

Our declassification mechanism – the flow declaration construct – can be made even more expressive. In particular it would be interesting to extend the language in order to have first-class security levels [Tse & Zdancewic, 2004; Zheng & Myers, 2004]. Moreover, the idea of using a construct for dynamically introducing flow policies can certainly be applied to various other programming paradigms. Conversely, one could think of restricting the usage of the flow declaration construct in some sensible ways, and adapt the non-disclosure policy accordingly.

Enforcement Mechanisms To enforce the security policies on the programs of our languages we have presented new type and effect systems that are motivated by rather similar principles. In particular, the one of Chapter 3 offers a variant of [Almeida Matos & Boudol, 2005] that restricts declassification to occur by means of declassification operations that are contained within a flow

[1] The choice of having references statically linked to threads, rather than to domains [Ravara *et al.*, 2003] is justified by the fact that the latter setting would not have raised the problem of the confidentiality of data that is transported along with threads. Furthermore, that setting can be mimicked in the one presented here, by attributing each of the domains' references to a thread that does not leave the corresponding domain, and by having other threads (that would not own any reference) move between domains.

declaration. We have thus highlighted the distinction between our new declassification paradigm and the more common declassification by value downgrading.

The type soundness proofs, which we explained in detail for the distributed language and for the non-disclosure policy, are also closely related. We have thus reasons to believe that the type soundness proof mechanism, which is based on the one given in [Almeida Matos & Boudol, 2005] (in turn related to the one for [Boudol & Castellani, 2002]), can be applied to other settings as well.

Some obvious topics for improvements of the type and effect system could be to incorporate polymorphism and type inference [Myers, 1999; Pottier & Simonet, 2003]. Further refinements could perhaps be obtained by considering a richer set of effects, including for instance the creation and deletion of references, the creation of threads, and more generally any action that modifies the context of an expression in the (abstract) machine that evaluates it.

5.2 Final Remarks

The Flow Declaration – Yet Another Declassification Mechanism?

As we have seen in the discussions on related work of Chapter 3 (see Section 3.5), proposals of declassification mechanisms abound in the literature. More than providing just another declassification mechanism, here we have suggested a way to face the "*challenge* [of] *determining what the nature of a downgrading mechanism should be and what kinds of security guarantees it permits*" [Zdancewic, 2004]. The key idea is that before thinking of how to control the usage of declassification, one should possess a good framework to express it. We believe that the declassification framework that was presented in this thesis is attractive for the following reasons:

- It provides a simple, yet flexible and powerful declassification mechanism – the flow declaration. In particular, it does not incorporate restrictions that go beyond the basic purpose of expressing declassification.

- It includes a security policy – non-disclosure – with satisfactory semantical properties. We point out two such properties that are among those suggested as "sanity checks" for security policies in [Sabelfeld & Sands, 2005]: *semantic consistency*, meaning that programs that are semantically equivalent are coherently classified as secure or insecure; *monotonicity of security*, meaning that, on one hand, non-disclosure is equivalent to non-interference for programs that do not use declassification, while on the other hand, programs do not become insecure when flow declarations are added to them.

- It is easily extendable to other languages and settings. In particular our non-disclosure property is *extensional*, i.e. defined in terms of program semantics, independently of the particularities of the language.

- It provides a sound technique for rejecting in a reasonably precise way all programs that do not satisfy the security property.

The first of the above merits is perhaps the strongest contribution of our declassification framework. Indeed, our flow declarations can express declassification

with any level of refinement, from specific operations to whole portions of a program, and between any security levels. This is achieved by directly manipulating flow policies, that are simply binary relations on the principals of a system.

Finally, we note that, by incorporating our declassification mechanism in our study of information flow control for networks, we have shown its robustness when used in new computation settings.

On Combining Declassification and Mobility

The topics of declassification and mobility in information flow are rather independent problems. It is perhaps not surprising that the two could be combined with little technical effort. However, we must point out that this facility is rooted in the highly decentralized nature of the flow declarations. No global agreement is assumed about the flow policies for declassification (as in [Mantel & Sands, 2004]). Moreover, the changes to the flow policy that are dynamically performed by programs have a local scope, and do not affect the whole system.

The potential dangers that are opened by allowing declassification in a mobile setting could perhaps seem to be more striking than its advantages. One can imagine the example of a migrating thread executing under a very permissive flow policy: once it arrives at a domain where another thread that owns secret references is computing, it can declassify that information, independently of the owners flow policy. This could be encoded in our language as follows:

$$d_1[(\text{goto } d_2); (\text{flow } H \prec L \text{ in } (m.b_L :=^{?} (?\ n.a_H)))^{m}] \ \parallel \ d_2[N^n] \qquad (5.1)$$

According to non-disclosure for networks, this program is secure – in fact, thread m complies with the declared flow policy when copying the value of the reference $n.a_H$ to $m.b_L$. However, one can see the potential of formulating other security policies that take into account the ownership of information, or of defining language constructs that condition the execution of subprograms to be executed under more strict flow policies, or even setting up of fire-wall-like conditions that control the entrance of mobile threads.

Dually, we find the example of a mobile thread that brings its own data to some site where it should perform private computations. Then, the possibility of declaring its own flow policies turns into an advantage. For instance, we could write the program:

$$d_1[(\text{goto } d_2); (\text{flow } H \prec L \text{ in } (n.b_L :=^{?} (?\ m.a_H)))^{m}] \ \parallel \ d_2[N^n] \qquad (5.2)$$

There is little practical experience in using mobile computing systems, which makes it hard to evaluate the particular relevance of allowing declassification in a mobile computing setting. Nevertheless, declassification seems to be a crucial feature in any language that is subject to information flow control, which *a fortiori* justifies the option of including it in the mobile language of Chapter 4. Moreover, in order to evaluate the problems or advantages that it might bring, it is favorable to study declassification on a simple, yet expressive, language, which can then be used as a starting point on which to build more complex frameworks. We believe that the mobile language presented in this thesis is a fertile starting ground for the study of secure information flow in networks.

References

Agat, J. 2000. Transforming out timing leaks. *In: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press.

Almeida Matos, A. 2005. Non-disclosure for distributed mobile code. *In: Foundations of Software Technology and Theoretical Computer Science.* Lecture Notes in Computer Science. To appear.

Almeida Matos, A. & Boudol, G. 2005. On declassification and the non-disclosure policy. *In: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05).* IEEE Computer Society.

Almeida Matos, A. & Boudol, G. & Castellani, I. 2004. Typing non-interference for reactive programs. *In: Proceedings of the Workshop on Foundations of Computer Security*, vol. 31. Turku Center for Computer Science.

Andrews, G. R. & Reitman, R. P. 1980. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, **2**(1), 56–76.

Banerjee, Anindya & Naumann, David A. 2005. Stack-based access control and secure information flow. *Journal of Functional Programming*, **15**(2), 131–177.

Bell, D. E. & La Padula, L. J. 1976. *Secure computer system: unified exposition and multics interpretation.* Technical Report MTR-2997. The MITRE Corporation.

Bossi, A. & Piazza, C. & Rossi, S. 2004. Modelling downgrading in information flow security. *In: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04).* IEEE Computer Society.

Boudol, G. 2004. ULM, a core programming model for global computing. *In: Programming Languages and Systems: 13th European Symposium on Programming.* Lecture Notes in Computer Science, vol. 2986. Springer-Verlag.

Boudol, G. 2005a. A generic membrane model. *In: Global Computing: IST/FET International Workshop.* Lecture Notes in Computer Science, vol. 3267. Springer-Verlag.

Boudol, G. 2005b. On typing information flow. *In: International Colloquium on Theoretical Aspects of Computing.* Lecture Notes in Computer Science, vol. 3722. Springer-Verlag.

Boudol, G. & Castellani, I. 2002. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, **281**(1), 109–130.

Boudol, G. & Castellani, I. & Germain, F. & Lacoste, M. 2002. *Analysis of formal models of distribution and mobility: state of the art.* Mikado Deliverable D1.1.1.

Boussinot, F. & Simone, R. 1996. The SL synchronous language. *Software Engineering*, **22**(4), 256–266.

Bugliesi, M. & Castagna, G. & Crafa, S. 2001. Boxed Ambients. *Lecture Notes in Computer Science*, **2215**, 38–63.

Cardelli, L. & Gordon, A. D. 2000. Mobile Ambients. *Theoretical Computer Science*, **240**(1), 177–213.

Chong, S. & Myers, A. C. 2004. Security policies for downgrading. *In: Proceedings of the 11th ACM conference on Computer and communications security.* ACM Press.

Clark, D. & Hunt, S. & Malacaria, P. 2004. Quantified interference: information theory and information flow. *In: Proceedings of the Workshop on Issues in the Theory of Security 2004.*

Cohen, E. 1977. Information transmission in computational systems. *In: Proceedings of the sixth ACM symposium on Operating systems principles.* ACM Press.

Crafa, S. & Bugliesi, M. & Castagna, G. 2002. Information flow security for Boxed Ambients. *In: International Workshop on Foundations of Wide Area Network Computing.* Electronic Notes in Theoretical Computer Science, vol. 66(63). Elsevier Science Publishers.

Crary, K. & Kliger, A. & Pfenning, F. 2005. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, **15**(02).

Denning, D. E. 1976. A lattice model of secure information flow. *Communications of the ACM*, **19**(5), 236–243.

Di Pierro, A. & Hankin, C. & Wiklicky, H. 2002. Approximate noninterference. *In: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02).* IEEE Computer Society.

Ferrari, E. & Samarati, P. & Bertino, E. & Jajodia, S. 1997. Providing flexibility in information flow control for object oriented systems. *In: Proceedings of the 1997 IEEE Symposium on Security and Privacy.* IEEE Computer Society.

Focardi, R. & Gorrieri, R. 1995. A classification of security properties for process algebras. *Journal of Computer Security*, **3**(1), 5–33.

Goguen, J. A. & Meseguer, J. 1982. Security policies and security models. *In: Proceedings of he 1992 IEEE Computer Society Symposium on Research in Security and Privacy.* IEEE Computer Society.

Heintze, N. & Riecke, J. G. 1998. The SLam calculus: programming with secrecy and integrity. *In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press.

Hennessy, M. & Riely, J. 2002. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, **24**(5), 566–591.

Hicks, M. & Tse, S. & Hicks, B. & Zdancewic, S. 2005. Dynamic updating of information-flow policies. *In: Proceedings of the International Workshop on Foundations of Computer Security (FCS).*

Honda, K. & Yoshida, N. 2002. A uniform type structure for secure information flow. *In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages POPL '02.* ACM Press.

Honda, K. & Vasconcelos, V. & Yoshida, N. 2000. Secure information flow as typed process behaviour. *In: Programming Languages and Systems: 9th European Symposium on Programming.* Lecture Notes in Computer Science, vol. 1782. Springer-Verlag.

Jones, A. K. & Lipton, R. J. 1975. The enforcement of security policies for computation. *In: Proceedings of the fifth ACM symposium on Operating systems principles.* ACM Press.

Kirli, D. 2000. Mobile functions and secure information flow. *In: Proceedings of the 27th International Colloquium on Automata, Languages, and Programming.*

Lampson, B. W. 1973. A note on the confinement problem. *Communications of the ACM*, **16**(10), 613–615.

Laud, P. 2001. Semantics and program analysis of computationally secure information flow. *In: Programming Languages and Systems: 10th European Symposium on Programming.* Lecture Notes in Computer Science, vol. 2028. Springer-Verlag.

Laud, P. 2003. Handling encryption in an analysis for secure information flow. *In: Programming Languages and Systems: 12th European Symposium on Programming.* Lecture Notes in Computer Science, vol. 2618. Springer-Verlag.

Li, P. & Zdancewic, S. 2005. Downgrading policies and relaxed noninterference. *In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press.

Li, P. & Mao, Y. & Zdancewic, S. 2003. Information integrity policies. *In: Proceedings of the First Workshop on Formal Aspects in Security and Trust (FAST).*

Lowe, G. 2004. Semantic models for information flow. *Theoretical Computer Science*, **315**(1), 209–256.

Lucassen, J. M. & Gifford, D. K. 1988. Polymorphic effect systems. *In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press.

Mantel, H. 2001. Information flow control and applications – bridging a gap. *In: Formal Methods for Increasing Software Productivity: International Symposium of Formal Methods Europe*. Lecture Notes in Computer Science, vol. 2021. Springer-Verlag.

Mantel, H. & Sabelfeld, A. 2004. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, **11**(4), 615–676.

Mantel, H. & Sands, D. 2004. Controlled declassification based on intransitive noninterference. *In: Programming Languages and Systems: Second Asian Symposium*. Lecture Notes in Computer Science, vol. 3302. Springer-Verlag.

Milner, R. & Tofte, M. & Harper, R. & MacQueen, D. 1997. *The definition of Standard ML (Revised)*. The MIT Press.

Myers, A. & Liskov, B. 1998. Complete, safe information flow with decentralized labels. *In: 19th IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society.

Myers, A. & Sabelfeld, A. & Zdancewic, S. 2004. Enforcing robust declassification. *In: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*. IEEE Computer Society.

Myers, A. C. & Liskov, B. 1997. A decentralized model for information flow control. *In: Proceedings of the sixteenth ACM symposium on Operating systems principles SOSP '97*. ACM Press.

Myers, A. C. & Liskov, B. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, **9**(4), 410–442.

Myers, Andrew C. 1999. JFlow: Practical mostly-static information flow control. *In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press.

Pottier, F. & Simonet, V. 2003. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, **25**(1), 117–158.

Ravara, A. & Matos, A. & Vasconcelos, V. T. & Lopes, L. 2003. Lexically scoping distribution: what you see is what you get. *In: FGC: Foundations of Global Computing*. Electronic Notes in Theoretical Computer Science, vol. 85(1). Elsevier Science Publishers.

Roscoe, A. W. & Goldsmith, M. H. 1999. What is intransitive noninterference? *In: Proceedings of the 1999 IEEE Computer Security Foundations Workshop*. IEEE Computer Society.

Rushby, J. 1992. *Noninterference, transitivity, and channel-control security policies.* Technical Report CSL-92-02. SRI.

Ryan, P. & McLean, J. & Millen, J. & Gligor, V. 2001. Non-interference: who needs it? *In: Proceedings of the 14th IEEE Workshop on Computer Security Foundations.* IEEE Computer Society.

Sabelfeld, A. 2001. The impact of synchronization on secure information flow in concurrent programs. *In: Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics.* Lecture Notes in Computer Science, vol. 2244. Springer-Verlag.

Sabelfeld, A. & Mantel, H. 2002. Static confidentiality enforcement for distributed programs. *In: Static Analysis : 9th International Symposium.* Lecture Notes in Computer Science, vol. 2477. Springer-Verlag.

Sabelfeld, A. & Myers, A. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, **21**(1).

Sabelfeld, A. & Myers, A. 2004. A model for delimited information release. *In: International Symposium on Software Security (ISSS'03).* Lecture Notes in Computer Science, vol. 3233. Springer-Verlag.

Sabelfeld, A. & Sands, D. 2000. Probabilistic noninterference for multi-threaded programs. *In: Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00).* IEEE Computer Society.

Sabelfeld, A. & Sands, D. 2005. Dimensions and principles of declassification. *In: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05).* IEEE Computer Society.

Sandhu, R. S. 1993. Lattice-based access control models. *Computer*, **26**(11), 9–19.

Sekiguchi, T. & Yonezawa, A. 1997. A calculus with code mobility. *In: Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS).* Chapman and Hall.

Simonet, V. 2003. *The Flow Caml System: documentation and user's manual.* Technical Report 0282. Institut National de Recherche en Informatique et en Automatique (INRIA).

Smith, G. 2001. A new type system for secure information flow. *In: Proceedings of the 14th IEEE Workshop on Computer Security Foundations.* IEEE Computer Society.

Smith, G. & Volpano, D. 1998. Secure information flow in a multi-threaded imperative language. *In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press.

Tse, S. & Zdancewic, S. 2004. Run-time principals in information-flow type systems. *In: IEEE Symposium on Security and Privacy.* IEEE Computer Society.

VOLPANO, D. 2000. Secure introduction of one-way functions. *In: Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00).* IEEE Computer Society.

VOLPANO, D. & SMITH, G. 1997. Eliminating covert flows with minimum typings. *In: Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97).* IEEE Computer Society.

VOLPANO, D. & SMITH, G. 1999. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, **7**(2-3).

VOLPANO, D. & SMITH, G. 2000. Verifying secrets and relative secrecy. *In: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press.

VOLPANO, D. & SMITH, G. & IRVINE, C. 1996. A sound type system for secure flow analysis. *Journal of Computer Security*, **4**(3), 167–187.

WRIGHT, A. K. & FELLEISEN, M. 1994. A syntactic approach to type soundness. *Inf. Comput.*, **115**(1), 38–94.

ZDANCEWIC, S. 2003. A type system for robust declassification. *In: Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics.*

ZDANCEWIC, S. 2004. Challenges for information-flow security. *In: 1st International Workshop on the Programming Language Interference and Dependence (PLID'04).*

ZDANCEWIC, S. & MYERS, A. C. 2002. Secure information flow via linear continuations. *Higher Order Symbol. Comput.*, **15**(2-3), 209–234.

ZDANCEWIC, S. & ZHENG, L. & NYSTROM, N. & MYERS, A. C. 2002. Secure program partitioning. *ACM Transactions on Computer Systems*, **20**(3), 283–328.

ZHENG, L. & MYERS, A. 2004. Dynamic security labels and noninterference. *In: Proceedings of the 2nd International Workshop on Formal Aspects in security and Trust (FAST).*

Index