Un Framework para el Análisis Formal de Modelos de Control de Acceso para Dispositivos Móviles Interactivos

Tesina de grado presentada
por

Juan Manuel Crespo
C-3962/4

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Av. Pellegrini 250, Rosario, República Argentina

Abril 2009

Supervisores

Carlos D. Luna Gustavo Betarte

Instituto de Computación, Facultad de Ingeniería

Universidad de la República

Julio Herrera y Reissig 565, Montevideo, Uruguay

Resumen

Los dispositivos portátiles tales como teléfonos celulares y asistentes personales de datos, permiten almacenar información confidencial y establecer comunicaciones con entidades externas. Generalmente, los usuarios pueden descargar e instalar nuevas aplicaciones de fuentes no confiables, que conviven junto con las instaladas por el fabricante del dispositivo o proveedor de servicios de comunicación. Ante este escenario, es importante garantizar la confidencialidad e integridad de los datos almacenados, así como la disponibilidad del servicio, aún cuando una aplicación maliciosa trate de hacer uso indebido de las funciones del dispositivo.

La plataforma Java Micro Edition (JME), una tecnología para desarrollo de software Java, provee el estándar Mobile Information Device Profile (MIDP) que facilita el desarrollo de aplicaciones y especifica un modelo de seguridad para el acceso controlado a recursos sensibles del dispositivo. El modelo está construido sobre la noción de dominio de protección, que puede ser concebido como un conjunto de permisos.

Un modelo alternativo ha sido propuesto, que extiende los permisos presentes en MIDP, introduciendo la noción de multiplicidad, y flexibilizando la forma en la que el usuario puede conceder a las aplicaciones que son utilizadas en el dispositivo, accesos a los recursos del mismo.

Esta tesina presenta un framework, formalizado utilizando el asistente de pruebas Coq, adecuado para la definición y comparación formal de políticas de control de acceso que pueden ser aplicadas por variantes de esos modelos de seguridad y para el análisis y prueba de propiedades de seguridad que éstas satisfacen. Las pruebas de algunas de estas propiedades son dadas y discutidas en el trabajo. Además, se provee una generalización que abstrae el concepto de modelo de control de accesos y se define un concepto de generalidad que permite compararlos formalmente.

Índice general

Resumen III

Contenidos III

Índice de figuras VI

1. Introducción **1**

1.1. Modelos de Seguridad para Dispositivos Móviles 3

1.1.1. El Modelo de Seguridad JME–MIDP 3

1.1.2. Un Modelo Alternativo 4

1.2. El Cálculo de Construcciones Inductivas 5

1.2.1. λ-cálculo Simplemente Tipado 5

1.2.2. Extensión al CIC 8

1.2.3. Isomorfismo de Curry-Howard 10

1.2.4. Coq 11

2. Un Framework para Modelado de Control de Accesos **13**

2.1. Notación utilizada 13

2.2. Permisos 15

2.3. Programas 18

2.4. Trazas 21

3. Propiedades del Framework **24**

3.1. Políticas de Concesión de Permisos 24

3.2. Relación entre Políticas de Concesión de Permisos 26

4. Generalización de Modelos de Permisos **29**

4.1. Abstracción de Modelos de Permisos 29

4.2. Noción de Generalidad de Modelos de Permisos 36

4.3. Homomorfismo entre los Modelos 39

5. Conclusiones y Trabajos Futuros **42**

Bibliografía **44**

Índice de figuras

2.1. Semántica de las Instrucciones 21

2.2. Ejemplo de Grafo de Control de Flujo 22

4.1. Diagrama Conmutativo que representa la Monotonía del Homomorfismo 37

4.2. Propiedad de Homomorfismos entre Estructura de Permisos 38

4.3. Diagrama que representa el Homomorfismo entre la Estructura de Permisos de MIDP y la del Modelo de [3] 40

Agradecimientos

A mis padres Alberto y Analía, por hacerme comprender desde muy pequeño el valor de la educación y por siempre apoyarme en todos mis proyectos.

A mi esposa Natalia, por su amor incondicional, su infinita paciencia y aliento durante estos últimos ocho años.

Al resto de mi familia, por su apoyo incondicional.

A mis compañeros, en especial a Damián, Julián, Germán y Pablo, por las innumerables charlas que hemos tenido a lo largo de estos años, que siempre han motivado mi apetito intelectual y mi amor por la ciencia.

A mis amigos de siempre, Nacho, Seba, Paulo, Santy, Vicky, Vale, Peyi y en especial al Colo, por compartir los momentos buenos y por bancarme en los malos.

A mis profesores, en especial a Guido, a Fidel y a Dante, por estar siempre dispuestos a contestar mis preguntas, por brindarme su apoyo, su consejo y su aliento en todos mis proyectos.

A mis supervisores, Carlos y Gustavo, por comprender mis urgencias y por estar siempre dispuestos a colaborar y a ayudarme con todos los problemas que me crucé por el camino.

A mis alumnos, por todo lo que aprendí de ellos y con ellos. Me gustaría decir que dando clases en las aulas de la facultad enseñé más de lo que aprendí, pero estaría mintiendo.

A la gente de IMDEA y CLIP, en especial a Gilles, por siempre asegurarse de que me encontrara cómodo y por hacerme sentir en casa cuando estaba muy lejos.

1

Introducción

Algunos dispositivos como Teléfonos Celulares o PDA's generalmente tienen acceso a información personal y están suscritos a servicios pagos para poder comunicarse con otras entidades. Además, los usuarios de dichos dispositivos pueden descargar e instalar aplicaciones desde fuentes no confiables a voluntad. En este contexto, el acceso que las aplicaciones tienen a los recursos del dispositivo es muy importante, ya que pueden por ejemplo, exponer información confidencial, hacer que el dispositivo utilice servicios pagos sin que el usuario este conciente de ello o simplemente destruir el dispositivo.

Java Micro Edition (JME) [15] es una versión de la plataforma Java diseñada para dispositivos con recursos acotados que comprende dos tipos de componentes: configuraciones y perfiles. El Mobile Information Device Profile (MIDP) [10, 11] define el ciclo de vida de una aplicación, un modelo de seguridad y API's que ofrecen la funcionalidad adecuada para el desarrollo de aplicaciones móviles, incluyendo networking, interfaces de usuario, push activation y almacenamiento persistente local.

Muchos fabricantes de dispositivos móviles han adoptado MIDP ya que su especificación está disponible. Una especificación formal del modelo de seguridad de JME-MIDP 2.0 fue desarrollada utilizando el asistente de pruebas Coq, descripta en detalle en [5].

En [3], se presenta un modelo de seguridad para dispositivos móviles interactivos. Éste puede ser visto como una extensión del modelo JME-MIDP. El trabajo presentado en ese artículo se enfoca en desarrollar un modelo formal para estudiar, en particular, mecanismos interactivos para la concesión de permisos para aplicaciones que se ejecutan en dispositivos móviles. La noción de permiso es central en este modelo, así como en MIDP, que es extendido

en dos sentidos: los permisos se generalizan con multiplicidades y se brinda más flexibilidad a la forma en la que el usuario otorga permisos a las aplicaciones.

Uno de los objetivos principales del presente trabajo ha sido construir un framework que provea un marco formal para definir y analizar modelos de permisos como MIDP o el presentado en [3]. Este framework, que está definido formalmente utilizando el Cálculo de Construcciones Inductivas [7, 8], adopta, con algunas variaciones, la mayoría de las construcciones de seguridad y de programación de [3]. La diferencia fundamental es que todas nuestras construcciones están parametrizadas por la política con la que se conceden los permisos. Se muestra como el framework puede ser utilizado para definir un tipo de "políticas de concesión de permisos" para luego definir los cuatro modos de permisos de MIDP, así como también las políticas definidas en [3], como objetos de ese tipo.

El trabajo también presenta una relación de orden entre estas políticas, basada en la noción de programa seguro, que puede ser utilizada para hacer un análisis comparativo de las mismas. En particular, se desarrolla una breve descripción de una prueba, construida utilizando el asistente de pruebas Coq [16], de un teorema que establece cómo las políticas mencionadas más arriba están relacionadas de acuerdo con este orden. Esta parte del trabajo ha sido publicado en [9].

Además, se realiza una generalización del concepto de modelo de control de acceso, abstrayendo la estructura de permisos subyacentes que éstos definen. Esta generalización permite, por un lado, expresar modelos de control de acceso en un marco formal unificado y por otro, caracterizar cuándo un modelo de control de accesos es más general que otro, es decir, cuando un modelo de control de accesos puede ser representado o simulado por otro. Esta caracterización es formal y esta basada en el concepto de homomorfismo. En particular, se define el homomorfismo entre la estructura de permisos de MIDP y la de [3] brindando evidencia formal de que el segundo es más general.

La definición completa del framework, así como los teoremas y sus pruebas formales están disponibles en: `www.fing.edu.uy/inco/grupos/mf/projects/PermModel/ACM-Coq.zip`.

La estructura del trabajo está organizada como se describe a continuación. En lo que resta de la introducción, se desarrolla una descripción de alto nivel de los modelos de seguridad en cuestión. Además, con el objetivo de hacer la presentación lo más auto-contenida posible, se describe brevemente el lenguaje formal utilizado para la especificación, el Cálculo de Construc-

ciones Inductivas. En el capítulo 2 se brinda una descripción formal del framework especificado. En el capítulo 3 se presentan los criterios para relacionar las políticas de concesión de permisos, así como la relación entre los modelos analizados. El capítulo 4 define los conceptos necesarios para la generalización y presenta una descripción detallada de la misma. El trabajo finaliza con conclusiones y trabajos futuros en el capítulo 5.

1.1. Modelos de Seguridad para Dispositivos Móviles

En esta sección se da una breve descripción informal de los modelos de permisos que son analizados en el presente trabajo.

1.1.1. El Modelo de Seguridad JME–MIDP

En MIDP, las aplicaciones (MIDlets) son empaquetadas y distribuidas en "suites". Una suite puede contener uno o mas MIDlets y es distribuida en dos archivos, un descriptor de aplicación y un archivo que contiene las clases y recursos. Una suite que requiere acceso a APIs protegidas o funciones tiene que solicitar los permisos correspondientes en su descriptor. Los suites pueden solicitar permisos en dos formas: requeridos u opcionales.

En la primera versión de MIDP [11], cualquier aplicación no instalada por el fabricante del dispositivo o el proveedor de servicios, corre en un "sandbox" que prohíbe el acceso a partes sensibles de las APIs o funciones del dispositivo. Este modelo de seguridad ha probado ser efectivo y previene cualquier posibilidad de que una aplicación comprometa la seguridad del dispositivo, al costo de ser excesivamente restrictivo y no permitir el desarrollo y distribución de aplicaciones útiles luego de la fabricación del dispositivo.

La versión 2.0 de MIDP [10] introduce un nuevo modelo de seguridad basado en el concepto de dominio de protección. Un dominio de protección puede ser visto como una abstracción del contexto de ejecución de una aplicación, y determina los derechos de acceso a funciones protegidas del dispositivo. Cada API o función del dispositivo "sensible" puede definir permisos para prevenir su uso no autorizado. Un dominio de protección comprende un conjunto de permisos que son otorgados incondicionalmente y sin la intervención del usuario del dispositivo (llamados permisos "allowed") así como un conjunto de permisos que requieren la autorización

explícita del usuario (llamados permisos “user”). Los permisos pueden ser concedidos por el usuario a una suite activa en algunos de estos tres modos:

- **blanket**: el permiso es concedido por el tiempo que la aplicación permanezca instalada en el dispositivo.
- **session**: el permiso es concedido por el tiempo que esté activa la aplicación.
- **one-shot**: el permiso es concedido para un solo uso.

Cada suite instalada está asociada a un único dominio de protección. Las suites no confiables están asociadas a un dominio de protección con permisos equivalentes a aquellos encontrados en el sandbox de MIDP 1.0. Las suites confiables pueden ser identificadas mediante un sistema de firmas criptográficas y entonces asociarse a dominios de protección mas permisivos. Este modelo permite que las aplicaciones desarrolladas por terceros confiables puedan ser descargadas e instaladas por los usuarios luego de la entrega del dispositivo, sin comprometer su seguridad.

El conjunto de permisos que son efectivamente otorgados a una suite es determinado por su dominio de protección, los permisos que la suite solicita en su descriptor y los permisos otorgados por el usuario.

Para una descripción mas detallada de los mecanismos definidos por este modelo de seguridad, se invita al lector a consultar [10, 11]. Una especificación formal del modelo de seguridad MIDP 2.0 es presentada en [5] y un controlador de accesos certificado para el “enforcement” de políticas permitidas por el modelo es descripto en detalle en [14].

1.1.2. Un Modelo Alternativo

En [3], se presenta un modelo de seguridad para dispositivos móviles que puede considerarse en algún sentido como una generalización del modelo MIDP. Ese trabajo se enfoca en desarrollar un modelo formal para estudiar, en particular, mecanismos interactivos para concesión de permisos por parte del usuario para aplicaciones que se ejecutan en dispositivos móviles. Como en el caso de MIDP, la noción de permiso es central en este modelo. Una generalización del permiso one-shot descripto arriba es propuesta, que consiste en asociar cada permiso con una multiplicidad que representa cuantas veces el permiso puede ser utilizado.

El modelo propuesto tiene dos instrucciones básicas para manipular permisos: **“grant”** y **“consume”**. La instrucción grant modela la consulta interactiva al usuario, en la cual se le pregunta si autoriza un permiso con cierta multiplicidad. La instrucción consume, modela el acceso a una función sensible protegida por la política de seguridad, y por lo tanto consume los permisos previos.

Así mismo, se propone una semántica para las construcciones del modelo, como también una lógica para razonar acerca de propiedades del flujo de ejecución de los programas. La garantía de seguridad básica que brinda la lógica, es que permite probar que un programa nunca intentará utilizar recursos para los que no posee permisos. Los autores también proporcionan un análisis estático que permite calcular una cota superior en los permisos iniciales necesarios para que un programa se ejecute sin violar dicha propiedad de seguridad. Para desarrollar dicho análisis las instrucciones son integradas en un modelo de programas basado en grafos de control de flujo. Este modelo también ha sido utilizado en trabajos previos acerca de modelado de control de accesos para Java, por ejemplo en [4, 12].

1.2. El Cálculo de Construcciones Inductivas

Para la especificación del framework se ha optado por utilizar el Cálculo de Construcciones Inductivas (CIC). Una de las razones principales que justifican esta elección es el hecho de que brinda un entorno unificado para la definición y verificación de formalizaciones. Además, la disponibilidad de herramientas de software simplifica mucho el desarrollo.

El Cálculo de Construcciones Inductivas [2] es un sistema formal que extiende al Cálculo de Construcciones (CoC) de Coquand y Huet [7, 8] con la noción primitiva de definición inductiva. CoC es en sí mismo, una extensión del cálculo lambda simplemente tipado presentado por Church[6].

1.2.1. λ-cálculo Simplemente Tipado

La mejor forma de introducir CIC es empezando por el λ-cálculo simplemente tipado (sin polimorfismo), para luego describir las extensiones propuestas por Coquand y Huet. A continuación se describen las principales características.

Tipos

Los tipos del λ-cálculo simplemente tipado pueden ser construidos a partir de tipos atómicos α, β, ... que pueden combinarse por medio del constructor $\rightarrow$ (flecha).

Más formalmente,

1. si A es un tipo atómico entonces es un tipo;

2. si A, B son tipos entonces $A \rightarrow B$ es un tipo.

El tipo $A \rightarrow B$ representa el espacio de funciones entre A y B, y es un caso particular de una construcción más general del CIC llamada producto. Es importante señalar que las funciones que consideramos son funciones totales y representan un proceso computacional que termina en tiempo finito.

Declaraciones y Definiciones

Una declaración se utiliza para asociar un tipo a un identificador, dejando subespecificado su valor. La declaración de un identificador foo de tipo A se nota $(x : A)$.

Una definición determina el valor de un identificador asociándolo con un término válido. Como éste término debe tener un tipo, una definición también asigna un tipo al identificador. Esta se nota como $(x : A := t)$.

El conjunto de declaraciones y definiciones en un momento dado se denomina contexto. En un contexto válido un identificador puede estar definido o declarado a lo sumo una vez.

Expresiones

Las expresiones se construyen a partir de variables y constantes utilizando abstracción y aplicación. Las expresiones válidas son aquellas que tienen tipo. A continuación daremos las reglas de tipado de las expresiones.

Tipado de Expresiones

- **Identificadores** Un identificador x es una expresión válida siempre que se encuentre declarado en el contexto actual Γ. La regla correspondiente es:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \ Var$$

- **Aplicación** Sean e_1 y e_2 son expresiones que en un contexto Γ tienen tipo $A \rightarrow B$ y A respectivamente. La aplicación de e_1 a e_2 se nota $(e_1\ e_2)$ y es un término de tipo B en el contexto Γ,

$$\frac{\Gamma \vdash e_1 : A \rightarrow B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash (e_1\ e_2) : B} \ App$$

- **Abstracción** Si en el contexto construido al agregar $(x : A)$ a Γ, e tiene tipo B, entonces en el contexto Γ, $(\lambda x.e)$ tiene tipo $A \rightarrow B$,

$$\frac{\Gamma :: (x : A) \vdash e : B}{\Gamma \vdash (\lambda x.e) : A \rightarrow B} \ Abs$$

Reducción

La evaluación de expresiones del lenguaje se realiza mediante reducción. Se consideran dos tipos de reducciones, la δ-reducción y la β-reducción.

- **δ-reducción** Se utiliza para reemplazar un identificador por la definición del mismo en el contexto actual. Si e_1 es un término en el que aparece x, y en el contexto actual x está definido como e_2, entonces e_1 se δ-reduce a $e_1\{x/e_2\}$, que denota la sustitución libre de capturas (α-convirtiendo cuando sea necesario) de las ocurrencias x por e_2 en e_1.

- **β-reducción** Hace posible reducir un término de la forma $(\lambda x.e1)e2$ al término $e_1 x/e_2$, donde, como en el caso anterior, la sustitución es libre de captura.

Convertibilidad

Diremos que dos términos t y t' son convertibles ($t =_{\delta\beta} t'$) si pueden ser reducidos al mismo término mediante una secuencia de δ y β-reducciones. La convertibilidad en λ-cálculo

simplemente tipado es decidible. Lo que es más, satisface la propiedad de normalización débil (todo término tiene forma normal) por lo que para decidir si dos términos son convertibles basta con reducirlos a su forma normal y comparar los términos obtenidos.

1.2.2. Extensión al CIC

CIC extiende el λ-cálculo simplemente tipado con una jerarquía de tipos y además, define una noción primitiva de tipo inductivo. Además, proporciona un operador de punto fijo para la definición de funciones recursivas. En éste formalismo se elimina la distinción sintáctica entre términos y tipos del lenguaje, los tipos del lenguaje son considerados también términos. Por lo tanto, en este formalismo podemos construir términos y tipos mutuamente dependientes.

Sorts

En CIC, todos los términos tienen tipos. Pero además, hemos dicho que los tipos son también términos del lenguaje. Por lo tanto, aunque suene inconsistente, los tipos también deben tener tipo. El tipo de un tipo es siempre una constante del lenguaje denominada sort. Existen dos sorts básicos: Set y Prop.

- Prop es el tipo de las proposiciones lógicas. Si $P : Prop$, entonces es una proposición lógica, y viéndola como un tipo, denota un conjunto de objetos que representan sus pruebas. Cualquier objeto $m : P$ es un testigo de la verdad de P.
- Set se utiliza para describir tipos de datos y especificaciones. Todo término de tipo Set puede pensarse como una especificación, y todo término cuyo tipo es una especificación, es un programa. Por ejemplo, el tipo $\text{nat} \to \text{nat} : Set$ representa una especificación de una función de los naturales en los naturales. Cualquier función entre naturales satisface dicha especificación. Veremos que es posible escribir especificaciones mucho más expresivas.

Un sort también es un término del lenguaje y como tal, debe tener un tipo. Asumir simplemente que tiene $Set : Set$ introduce inconsistencias en la teoría, por lo tanto se introduce una jerarquía de universos formados por la familia de tipos $Type(i)$, con $i \in \mathbb{N}$, que satisface las siguientes

propiedades:

$$\begin{aligned} Set &: Type(0) \\ Prop &: Type(0) \\ Type(i) &: Type(i+1) \end{aligned}$$

En resumen, los sorts Set y Prop posibilitan la coexistencia de programas y especificaciones por un lado, y de pruebas y proposiciones por el otro. Prop se utiliza para codificar proposiciones y pruebas, mientras que Set se utiliza para representar especificaciones y programas. La jerarquía de tipos $Type(i)$ se introduce para tipar términos que son tipos, y conservar la disciplina de tipos del lenguaje.

Términos

Los términos de CIC se construyen a partir de constantes y variables utilizando los mecanismos de abstracción, aplicación, declaración local, y producto de acuerdo a las siguientes reglas:

1. Set, $Prop$ y $Type$ son términos;
2. una constante es un término;
3. una variable es un término;
4. si x es una variable y M, N son términos, entonces la λ-abstracción $(\lambda x : M.N)$ es un término en donde las ocurrencias de x en N están ligadas por la abstracción;
5. si x es una variable y M, N son términos, entonces la λ-abstracción $(\Pi x : M.N)$ es un término en donde las ocurrencias de x en N están ligadas por el producto;
6. si M y N son términos, entonces la aplicación (MN) es un término;
7. si x es una variable y M, N son términos, entonces $let\ x := M\ in\ N$ es un término que denota el término N donde las ocurrencias de x están ligadas localmente al término M.

Las constantes se introducen mediante declaraciones, o mediante definiciones de tipos inductivos (tanto el nombre del tipo como sus constructores son constantes). El producto permite

introducir tipos dependientes: en $(\Pi x : M, N)$, N puede depender de x. Cuando x no ocurre libre en N , puede escribirse simplemente como $M \rightarrow N$, que no es más que el tipo funcional del λ-cálculo simplemente tipado.

Reducción y Convertibilidad

La noción de reducción se extiende para tratar con tipos inductivos (ι-reducción) y definiciones locales (ζ-reducción). Intuitivamente, la ι-reducción reduce de la forma esperada la aplicación de un destructor a un término de un tipo inductivo, mientras que la ζ-reducción reemplaza las ocurrencias de variables en el cuerpo de una definición local por el valor al que están ligadas.

La relación de convertibilidad, definida como una relación de equivalencia en el λ-cálculo simplemente tipado, se extiende a una relación de orden $\leq_{\beta\delta\iota\zeta}$ que tiene en cuenta los sorts y la jerarquía de universos $Type(i)$ de CIC. El mecanismo de verificación e inferencia de tipos incorpora una regla que subsume los objetos de un tipo T en el tipo U siempre que $T \leq_{\beta\delta\iota\zeta} U$.

1.2.3. Isomorfismo de Curry-Howard

El isomorfismo de Curry-Howard, que evidencia la correspondencia entre el λ-cálculo simplemente tipado y las pruebas al estilo deducción natural de la lógica proposicional intuicionista, puede extenderse fácilmente a CIC. Si se interpreta un tipo como a una proposición, pueden interpretarse los términos de ese tipo como pruebas de la proposición. El poder de expresividad del sistema de tipos permite construir de esta forma objetos de prueba para proposiciones del cálculo de predicados intuicionista; la verificación de pruebas se reduce entonces a la verificación de tipos. El término $A \rightarrow B$ puede interpretarse como la proposición A implica B, o como el tipo de las funciones que van de A en B, pues una prueba de la proposición A implica B no es más que un procedimiento para obtener una prueba de B a partir de una prueba de A.

Consideremos, por ejemplo, el constructor de tipos $\rightarrow$, y asumamos que nat es el tipo de los números naturales. El constructor $\rightarrow$ puede utilizarse tanto para denotar el tipo de las funciones de los naturales en los naturales, $nat \rightarrow nat$, como para denotar el tipo de los predicados sobre números naturales, $nat \rightarrow Prop$. La misma λ-abstracción que se utiliza para construir funciones ordinarias como $(\lambda n : nat.n + 1)$, sirve también para construir predicados

sobre los números naturales, como $(\lambda n : nat.n = 0)$. Es más, si P tiene tipo $nat \rightarrow Prop$, entonces (Pn) es una proposición y $(\Pi n : nat, Pn)$ representa el tipo de las funciones que asocian a cada natural n, un objeto de tipo (Pn). Los objetos de $(\Pi n : nat, Pn)$ representan, por lo tanto, pruebas de la proposición $(\forall n.Pn)$.

Gracias al isomorfismo de Curry-Howard, y a la no distinción entre términos y tipos, puede utilizarse CIC tanto para construir programas como para razonar sobre sus propiedades. Aún mejor, pueden aprovecharse técnicas de razonamiento para construir programas, o técnicas de construcción de programas para construir pruebas.

1.2.4. Coq

El sistema Coq es un asistente de pruebas basado en CIC diseñado para el desarrollo de pruebas matemáticas y la verificación formal de programas. Coq provee un lenguaje de especificación llamado Gallina que extiende CIC, y entre otras cosas proporciona un mecanismo de secciones y módulos para estructurar convenientemente las teorías. Aunque no lo utilizaremos durante el tratamiento que haremos en este trabajo, el sistema de tipos de Coq permite definir tipos coinductivos cuyos términos son objetos infinito. El sistema viene acompañado de una biblioteca estándar de teorías entre las que se destacan teorías sobre el cálculo de predicados, aritmética natural y entera, listas definidas inductivamente, y una axiomatización de los números reales.

Construcción de Pruebas

Utilizando el isomorfismo de Curry-Howard, tanto los programas como sus propiedades y las pruebas de éstas, pueden ser representados uniformemente en el mismo lenguaje. Un usuario de Coq que desarrolla una teoría en el lenguaje, puede enunciar un teorema simplemente como un tipo del lenguaje y demostrarlo construyendo un término de dicho tipo. La existencia de un término del tipo garantiza que el tipo no es vacío y que el teorema, visto como una proposición lógica según el isomorfismo de Curry-Howard, es verdadero. El núcleo del sistema es un verificador de tipos de CIC que verifica la corrección de los objetos de prueba construidos.

Como asistente de pruebas, Coq proporciona mecanismos interactivos para la construcción de pruebas por medio de reglas al estilo de las reglas de deducción natural. Este mecanismo se

basa en una batería de tácticas, que evita la complejidad de tener que manipular directamente términos del lenguaje. Algunas de estas tácticas permiten construir pruebas automáticamente en casos donde el problema es decidible (e.g. tautologías en el cálculo proposicional intuicionista, aritmética de Presburger), o utilizando heurísticas en el caso de que no lo sea.

Extracción de Programas

La distinción entre programas y pruebas a nivel de tipos de CIC, en base a los sorts Set y Prop, hace posible la extracción del componente computacional de los programas especificados y verificados en Coq. El mecanismo de extracción permite obtener programas funcionales en los lenguajes OCaml, Haskell y Scheme.

El Sistema de Módulos de Coq

Coq provee un sistema de módulos muy expresivo basado en el sistema de módulos de Objective Caml. Los principales conceptos son módulos, signaturas y functores:

- La principal motivación para tener módulos es poder empaquetar definiciones que están relacionadas y forzar un sistema de nombres consistente para éstas.
- Las signaturas proveen interfaces para los módulos. Una signatura especifica que componentes de la estructura son visibles desde fuera de ella y con que tipo. Pueden ser utilizadas para esconder algunas componentes de módulos o para exportar componentes con un tipo restringido.
- Los functores son "funciones" entre módulos. Son utilizados para expresar estructuras parametrizadas: una estructura A parametrizada por una estructura B es simplemente un functor F con parámetro formal B (con una signatura especificada para B) que retorna una estructura A.

Para más información sobre Coq, se recomienda [2].

2

Un Framework para Modelado de Control de Accesos

En este capítulo se introducen las construcciones formales utilizadas para definir los conceptos de seguridad que constituyen el núcleo de ciertos mecanismos de control de accesos, para luego describir como esos mecanismos son utilizados para definir los modelos de concesión de permisos que son el objeto de este trabajo.

2.1. Notación utilizada

En esta sección brindamos una breve descripción de la notación utilizada. Si bien la formalización se ha realizado en Coq, que define su propia notación, creemos que ésta no es del todo apropiada ya que el lector no habituado al formalismo le puede llegar a resultar confusa, por lo tanto hemos utilizado una notación más familiar y amena.

Para igualdad y conectivos lógicos utilizamos notación estándar ($\wedge, \vee, \neg, \rightarrow, \forall, \exists$).

Funciones y predicados anónimos utilizan notación lambda (e.g. $\lambda\ (x : T)\ .\ x, \lambda\ (x : nat)\ .\ x > 10$).

En caso de que haya más de una abstracción se utiliza abreviación estándar para λ-términos, (e.g. $\lambda\ (x : nat)\ .\ \lambda\ (y : nat)\ .\ x + y \equiv \lambda\ (x : nat)\ (y : nat)\ .\ x + y$.

Una relación inductiva I es definida dando reglas de introducción de la forma:

$$\frac{P_1 \dots P_m}{I\ x_1 \dots x_n}$$

donde las variables que ocurren libres están implícitamente cuantificadas universalmente. De la misma manera, los tipos inductivos se definen dando sus constructores de la siguiente forma:

$$\begin{aligned} T \stackrel{def}{=} \quad & \mid C_1 : A_{1,1} \to \dots A_{1,n_1} \to T \\ & \vdots \\ & \mid C_m : A_{m,1} \to \dots A_{m,n_m} \to T \end{aligned}$$

donde $C_1 \dots C_n$ son los constructores de T.

Un tipo récord R se define de la siguiente manera:

$$R \stackrel{def}{=} \{field_1 : A_1, \dots, field_n : A_n\}$$

Esta definición genera un tipo inductivo no recursivo con un solo constructor $mkR : A_1 \to \dots A_n \to R$ y funciones de proyección $field_i : R \to A_i$. La aplicación de las proyecciones se abrevia utilizando notación punto: $field_i\, r = r.field_i$. Cuando a partir del contexto se entiende el tipo se escribe $\langle x_1, \dots, x_n \rangle$ en lugar de $mkR\ x_1, \dots, x_n$.

En la formalización desarrollada se han utilizado tipos inductivos que tienen formas válidas e inválidas de ser construidos. En el resto de este trabajo se adopta la convención de que un tipo con el mismo nombre, pero al que se le prefija *valid* es el tipo que consiste de sólo los constructores válidos del tipo original. Cada vez que se utilice esta convención estará claro cuales son los constructores válidos para el tipo en cuestión.

Los siguientes tipos inductivos paramétricos se asumen predefinidos:

- *option* T con constructores $None$: *option* T y *Some* : $T \to$ *option* T,
- listas finitas sobre T, *list* T. La lista vacía se denota $[]$ y el constructor infijo que inserta un elemento x en frente de una lista xs se denota $x \triangleright xs$. Listas snoc finitas sobre T, *snocList* T, que no son más que listas construidas insertando elementos al final también se utilizan. De nuevo $[]$ denota la lista snoc vacía y $xs \triangleleft\ a$ denota la inserción de un elemento x al final de la lista xs.

2.2. Permisos

A cada recurso controlado del dispositivo se le asigna un tipo. Definimos $ResType$ como el conjunto de tipos de recursos. Si $rt : ResType$ es un tipo de recurso, entonces $Resources\ rt$ y $Actions\ rt$ representan el conjunto de recursos de tipo rt disponibles en el dispositivo y el conjunto de acciones posibles sobre esos recursos respectivamente. Los permisos sobre un tipo de recurso están definidos de la siguiente forma:

$$
\begin{array}{lll}
PermRes & & (rt : ResType) \stackrel{def}{=} \\
& | & valid : list\ (Resources\ rt) \to list\ (Actions\ rt) \to PermRes\ rt \\
& | & invalid : PermRes\ rt
\end{array}
$$

Es decir, dado un tipo de recurso rt un objeto de tipo $PermRes\ rt$ es un conjunto (representado como una lista) de acciones y recursos de tipo rt o la constante $invalid$. Una relación $\sqsubseteq_{PermRes}$ es definida al aplicar inclusión de conjuntos componente a componente. Esta relación define una estructura de reticulado (lattice) donde $invalid$ es el elemento "bottom" $\bot_{PermRes}$ y $\sqcup_{PermRes}$ es un operador de menor cota superior (least upper bound) obtenido al aplicar la unión componente a componente.

Como fue mencionado anteriormente, una noción de multiplicidad de concesión de permisos se introduce en [3]. Las multiplicidades se construyen de tres maneras. Una multiplicidad es o bien un numero natural, o un valor especial ∞ que denota permisos irrestrictos o un valor de error $\bot$. Un tipo Mul representándolas se introduce mediante la siguiente definición inductiva:

$$
\begin{array}{ll}
Mul \stackrel{def}{=} & | \bot\ : Mul \\
& | \, val : nat \to Mul \\
& | \infty\ : Mul
\end{array}
$$

Es intuitivo ver que un nuevo reticulado se puede construir sobre Mul con $\bot$ y ∞ como los elementos "bottom" y "top" respectivamente. Las definiciones de extensiones obvias de funciones y predicados definidos sobre naturales a funciones y predicados sobre Mul, como $\sqsubseteq_{Mul}$, $+_{Mul}$ y $pred_{Mul}$, se dan a continuación.

$$\begin{aligned}\sqsubseteq_{Mul}: Mul \to Mul \to bool \quad \stackrel{def}{=} \quad & \lambda\, m\, m'\ .\ match\ (m,\ m')\ with \\ & |\bot, _ \Rightarrow true \\ & |_, \infty \Rightarrow true \\ & |val\ n, val\ n' \Rightarrow n \leq n' \\ & |_, _ \Rightarrow false \\ & end.\end{aligned}$$

$$\begin{aligned}+_{Mul} : Mul \to Mul \to Mul \quad \stackrel{def}{=} \quad & \lambda\, m\, m'\ .\ match\ (m,\ m')\ with \\ & |\bot, v \Rightarrow v \\ & |v, \bot \Rightarrow v \\ & |\infty, _ \Rightarrow \infty \\ & |_, \infty \Rightarrow \infty \\ & |val\ p, val\ q \Rightarrow val\ (p+q) \\ & end.\end{aligned}$$

$$\begin{aligned}pred_{Mul} : Mul \to Mul \quad \stackrel{def}{=} \quad & \lambda\, m\ .\ match\ m\ with \\ & |\bot \Rightarrow \bot \\ & |val\ n \Rightarrow pred\ n \\ & |\infty \Rightarrow \infty \\ & end.\end{aligned}$$

Los permisos acumulados sobre un tipo de recurso está formado por dos componentes: el conjunto de recursos y acciones permitidas y una multiplicidad. Un permiso de este tipo, para un tipo de recurso rt es modelado como objeto del siguiente tipo récord:

$$PermMul\ (rt : ResType) \stackrel{def}{=} \{permRes : PermRes\ rt; mul : Mul\}$$

El reticulado de permisos de un tipo de recurso rt se obtiene al definir el orden $\sqsubseteq_{PermMul}$ de la siguiente manera:

$$\begin{aligned}pm_1 \sqsubseteq_{PermMul} pm_2 \quad \stackrel{def}{=} \quad & pm_1.permRes \sqsubseteq_{PermRes} pm_2.permRes \\ \wedge \quad & pm_1.mul \sqsubseteq_{Mul} pm_2.mul\end{aligned}$$

donde pm_1 y pm_2 son objetos de tipo $PermMul\ rt$.

Ahora estamos en condiciones de definir el estado de permisos del dispositivo. El estado del dispositivo se representa como un mapeo que asocia un permiso a cada tipo de recurso. Por lo

tanto se define como el siguiente tipo dependiente funcional:

$$Perm \stackrel{def}{=} \forall (rt : ResType), PermMul\ rt$$

Decimos que dos permisos p_1 y p_2 son extensionalmente iguales si para cada tipo de recurso rt vale $p_1\ rt\ = p_2\ rt$.

Un orden $\sqsubseteq_{Perm}$ puede ser definido como la extensión de producto de $\sqsubseteq_{PermMul}$ como sigue:

$$p_1 \sqsubseteq_{Perm} p_2 \stackrel{def}{=} \forall (rt : ResType), (p_1\ rt) \sqsubseteq_{PermMul} (p_2\ rt)$$

Para modelar las operaciones que afectan el estado de los permisos, una función *update* es definida:

$$update\ (p : Perm)(rt : ResType)(pres : PermRes\ rt)(m : Mul) : Perm$$

La definición de *update* es algo complicada, ya que utiliza el hecho de que la igualdad es decidible para $ResType$. El comportamiento esperado (y formalizado) de esta función es el usual de una función de actualización de memoria: el estado de los permisos no cambia para cualquier tipo de recurso distinto de rt, y para rt se actualiza a $\langle pres, m \rangle$.

Este comportamiento es capturado en los dos siguientes lemas:

Lemma 2.1.

$$\begin{aligned}
&Lemma\ updateDef_1 : \\
&\qquad \forall (p : Perm)(rt : ResType)(pres : PermRes\ rt)(m : Mul), \\
&\qquad (update\ p\ rt\ pres\ m)\ rt = \langle pres, m \rangle.
\end{aligned}$$

Lemma 2.2.

$$\begin{aligned}
&Lemma\ updateDef_2 : \\
&\qquad \forall (p : Perm)(rtrt' : ResType)(pres : PermRes\ rt)(m : Mul), \\
&\qquad rt <> rt' \rightarrow (update\ p\ rt\ pres\ m)\ rt' = p\ rt'
\end{aligned}$$

Estos lemas nos permiten abstraernos de su definición al realizar pruebas que involucran actualización de permisos.

Si rt es un tipo de recurso y p es un estado de los permisos, entonces la relación inductiva $Error$ se define de la siguiente manera:

$$\frac{(p\ rt).permRes = invalid\ rt}{Error\ p} \qquad \frac{(p\ rt).mul = \bot}{Error\ p}$$

La intuición es que una situación de error puede ocurrir cuando se intenta realizar una acción sobre un recurso de tipo rt y no hay permisos válidos asociados al recurso (primera regla), o cuando no hay permisos concedidos para ese recurso (segunda regla).

2.3. Programas

Un programa es representado por un grafo de control de flujo (CFG). Esta representación es muy común y se adopta, entre otros trabajos, en [1, 3]. El grafo de control de flujo captura las manipulaciones de permisos y el manejo de llamadas a métodos, retornos y excepciones.

Un grafo de control de flujo es una tupla $G = (NO, EX, KD, TG, CG, EG, n_0)$ donde:

- NO es el conjunto de nodos del grafo (uno por cada instrucción),
- EX es el conjunto de excepciones posibles,
- KD es una función de tipo $KD : NO \rightarrow Instr$ que asocia una instrucción a cada nodo,
- $TG : NO \rightarrow NO \rightarrow Prop$ es la función proposicional que caracteriza el conjunto de aristas intra-procedimentales (i.e. $n_1\ TG\ n_2$ si el control se puede transferir de una instrucción en el nodo n_1 a la instrucción correspondiente al nodo n_2 dentro del procedimiento actual),
- CG es el conjunto de aristas ínter-procedimentales (que puede ser utilizado para capturar llamadas a métodos dinámicos),
- $EG : EX \rightarrow NO \rightarrow NO \rightarrow Prop$ es la función proposicional que captura las aristas de excepción intra-procedimental,
- $n_0 : NO$ es el nodo inicial del grafo, correspondiente a la instrucción inicial del programa.

Las instrucciones son formalmente definidas en el framework mediante un tipo inductivo:

$$
\begin{aligned}
Instr \stackrel{def}{=} & \\
\mid \; & Grant : \forall (rt : ResType), validPermRes\ rt \rightarrow MulValid \rightarrow Instr \\
\mid \; & Consume : \forall (rt : ResType), validPermRes\ rt \rightarrow Instr \\
\mid \; & Call : Instr \\
\mid \; & Return : Instr \\
\mid \; & Throw : EX \rightarrow Instr
\end{aligned}
$$

donde $MulValid$ es el tipo de multiplicidades válidas (i.e. multiplicidades distintas de $\bot$).

La semántica operacional de los programas depende fuertemente de los mecanismos con los que se conceden y consumen los permisos. Estos mecanismos se describen y se discuten brevemente a continuación.

En [3] se discuten dos variantes acerca del efecto de la actualización luego de la concesión de un permiso: o bien los permisos antes de la concesión se descartan o se acumulan. A primera vista estas políticas de concesión de permisos tienen ventajas y desventajas. Independientemente de esa discusión particular, en este punto el modelo propuesto por los autores introduce una generalización con respecto al modelo MIDP: los permisos tienen multiplicidades asociadas. Uno de los objetivos principales del presente trabajo ha sido diseñar un framework que provea un marco teórico uniforme donde esos modelos de permisos puedan ser definidos formalmente y comparados. Con ese objetivo, las construcciones definidas para proporcionar semántica al comportamiento computacional de los programas, así como razonar sobre él han sido parametrizadas por la política de concesión de permisos. Estas políticas son capturadas formalmente por objetos del tipo siguiente:

$$
\begin{aligned}
grantPolicy \stackrel{def}{=} \; & \forall (rt : ResType), \\
& validPermRes\ rt \rightarrow NZMulValid \rightarrow Perm \rightarrow Perm
\end{aligned}
$$

donde un objeto de tipo $NZMulValid$ es una multiplicidad valida construida a partir de un natural distinto de cero.

En lo que respecta a la operación de consumo de permisos, su definición formal se detalla

a continuación:

$$\begin{aligned} consume \quad & (rt : ResType)(pr : validPermRes\ rt)(p : Perm) : Perm \stackrel{def}{=} \\ & if\ (pr\ \sqsubseteq_{PermRes} (p\ rt).permRes) \\ & then\ update\ p\ rt\ (p\ rt).permRes\ (pred_{Mul}\ (p\ rt).mul) \\ & else\ update\ p\ rt\ (invalid\ rt)\ (pred_{Mul}\ (p\ rt).mul) \end{aligned}$$

La idea intuitiva es que al intentar consumir permisos sobre un conjunto de recursos de tipo rt, se chequea si se posee permisos para ese conjunto de recursos. En caso afirmativo, los permisos se actualizan haciendo decrecer la multiplicidad asociada a ese tipo de recurso. Caso contrario, se produce una situación de error, que se ve reflejada en el hecho de que se actualiza el conjunto asociado a rt a $invalid$.

La operación de consumo es monótona respecto al orden entre permisos. Esto se explicita en el siguiente lema:

Lemma 2.3.

$$\begin{aligned} & Lemma\ consumeMon : \\ & \quad \forall (rt : ResType)(pr : validPermRes\ rt)(p\ p' : Perm), \\ & \quad p \sqsubseteq_{Perm} p' \rightarrow (consume\ rt\ pr\ p) \sqsubseteq_{Perm} (consume\ rt\ pr\ p') \end{aligned}$$

Siguiendo el enfoque de [3], la semántica operacional small-step de un grafo de control de flujo se ha definido como una relación que define transiciones entre estados (que consisten de un stack de nodos del grafo, enriquecidos con los permisos con los que se cuenta en ese punto de la ejecución). Esta definición ha sido extendida, al hacerla depender de la política de concesión de permisos. Formalmente se ha definido como una función proposicional inductiva $\leadsto_g$ cuyas reglas aparecen en la Fig. 2.1. Una propiedad importante de esta semántica es que es "non-intrusive", lo que significa que el estado de los permisos no interfiere con la ejecución. En otras palabras, una transición no se bloquea por ausencia de permisos. Esto se establece formalmente en el siguiente lema:

Lemma 2.4.

$$\begin{aligned} & Lemma\ nonIntrusive : \\ & \quad \forall (g : grantPolicy)(s\ s' : list\ NO)(ex\ ex' : option\ EX)(p\ p' : Perm), \\ & \quad s\ ex\ p\ \leadsto_g s' ex' p' \rightarrow \forall (p : Perm), (\exists (p' : Perm), s\ ex\ p\ \leadsto_g\ s'\ ex'\ p') \end{aligned}$$

b

$$\frac{KD\ n = Grant\ rt\ pr\ m \quad TG\ n\ n'}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n' \triangleright s)\ None\ (g\ rt\ pr\ m\ p)}$$

$$\frac{KD\ n = Consume\ rt\ pr \quad TG\ n\ n'}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n' \triangleright s)\ None\ (consume\ rt\ pr\ p)}$$

$$\frac{KD\ n = Call \quad CG\ n\ n'}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n' \triangleright n \triangleright s)\ None\ p} \qquad \frac{KD\ r = Return \quad TG\ n\ n'}{(r \triangleright n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n' \triangleright s)\ None\ p}$$

$$\frac{KD\ n = Throw\ ex \quad EG\ ex\ n\ h}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (h \triangleright s)\ None\ p} \qquad \frac{KD\ n = Throw\ ex \quad \forall (h : NO), \neg EG\ ex\ n\ h}{(n \triangleright s)\ None\ p\ \rightsquigarrow_g\ (n \triangleright s)\ (Some\ ex)\ p}$$

$$\frac{\forall (h : NO), \neg EG\ ex\ n\ h}{(t \triangleright n \triangleright s)\ (Some\ ex)\ p\ \rightsquigarrow_g\ (n \triangleright s)\ (Some\ ex)\ p} \qquad \frac{EG\ ex\ n\ h}{(t \triangleright n \triangleright s)\ (Some\ ex)\ p\ \rightsquigarrow_g\ (h \triangleright s)\ None\ p}$$

Figura 2.1: Semántica de las Instrucciones

2.4. Trazas

En [3] los resultados globales de la ejecución de programas se modelan utilizando trazas, que son definidos a partir de la semántica operacional definida más arriba (instanciada para una política de concesión de permisos particular) como sigue: *una traza parcial de un grafo de control de flujo es una secuencia (de tipo snocList $(NO, option\ EX)$) de nodos $[] \triangleleft \langle n_0, None\rangle \triangleleft \langle n_1, e_1\rangle \triangleleft \cdots \triangleleft \langle n_k, e_k\rangle$ tal que para todo $0 \leq\ i < k$ existe $\rho, \rho' \epsilon\ Perm$, $s, s' \epsilon\ (list\ NO)$ y verificando $n_i \triangleright s, e_i, \rho \rightsquigarrow n_{i+1} \triangleright s', e_{i+1}, \rho'$.*

Los stacks s y s' en la definición anterior aparecen cuantificados existencialmente porque no son definidos como componentes de las trazas. Esta cuantificación induce una pérdida de información con respecto a la semántica operacional. Un ejemplo [1] clarificará la situación. Consideremos el siguiente CFG:

$$NO = \{A, B, C, D\},\ TG = \{(B, C), (C, D)\},\ EX = CG = EG = \{\},\ n_0 = A$$
$$KD = \{(A, Return), (B, x), (C, Consume\ rt\ y), (D, Return)\}$$

donde $x : Instr$, $rt : ResType$, $y : validPermRes\, rt$, y con estado inicial de permisos $p_{init} = \lambda\ (rt : ResType)\ .\ \langle (valid\ rt\ []\ []), (val\ 0)\rangle$. La Fig. 2.2 muestra el CFG en cuestión. En este ejemplo se puede notar que $[] \triangleleft \langle A, None\rangle$ es la única traza admisible que arroja un estado de permisos válido. De acuerdo con la definición de traza parcial dada más arriba, el objeto $([] \triangleleft \langle A, None\rangle \triangleleft \langle C, None\rangle \triangleleft \langle D, None\rangle)$ es admitido como una traza parcial del CFG.

[1] Se agradece a Santiago Zanella por este ejemplo.

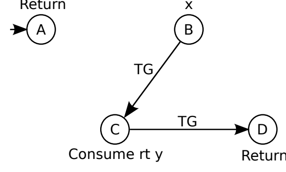

Figura 2.2: Ejemplo de Grafo de Control de Flujo

Esta traza puede ser construida utilizando las reglas de transición correspondientes a las instrucciones *Consume* y *Return* (ver Fig. 2.1). Sin embargo, la traza arroja una situación de error porque la transición del nodo C al D intenta consumir un permiso no disponible.

La definición de traza de ejecución de programas que ha sido propuesta en el framework presentado remedia la situación descripta arriba al incluir el nodo del stack como una componente de los elementos de la traza. Esto se evidencia en el tipo siguiente:

$$Trace \stackrel{def}{=} snocList\ \{noT : NO, stT : list\ NO, exT : option\ EX\}$$

La noción de traza parcial parametrizada se define inductivamente sobre elementos del tipo $Trace$ como sigue:

$$\frac{}{PTrace_g\ []} \qquad \frac{}{PTrace_g([] \triangleleft \langle n_0, [], None\rangle)}$$

$$\frac{PTrace_g\ (tr \triangleleft \langle n, s, ex\rangle) \quad \exists (p\ p' : Perm), n \triangleright s\ ex\ p \rightsquigarrow_g n' \triangleright s'\ ex'\ p'}{PTrace_g\ (tr \triangleleft \langle n, s, ex\rangle \triangleleft \langle n', s', ex'\rangle)}$$

Sea tr una traza y g una política de concesión de permisos. si $PTrace_g\ tr$ vale, entonces diremos que tr es una traza válida de acuerdo a g.

Dada una traza tr y una política g, la función $PermsOf_g : Perm \rightarrow Trace \rightarrow Perm$ computa el estado de los permisos resultante de la ejecución de un programa que arroja traza tr:

$$\begin{array}{l}
PermsOf_g(p_{init} : Perm)(tr : Trace) : Perm \stackrel{def}{=} \\
match\ tr\ with \\
\quad |[] \Rightarrow p_{init} \\
\quad |tr' \triangleleft e \Rightarrow match\ KD\ e.noT\ with \\
\qquad |Consume\ rt\ pr \Rightarrow consume\ rt\ pr\ (PermsOf_g\ p_{init}\ tr') \\
\qquad |Grant\ rt\ pr\ m \Rightarrow g\ rt\ pr\ m\ (PermsOf_g\ p_{init}\ tr') \\
\qquad |_ \Rightarrow PermsOf_g\ p_{init}\ tr' \\
\quad end \\
end
\end{array}$$

Finalmente, dada una política g, una traza es segura si ninguno de sus prefijos arroja un estado de permisos erróneo:

$$Safe_g(tr : Trace)(p_{init} : Perm) \stackrel{def}{=} \\ \forall tr' : Trace, (prefix\ tr'\ tr) \rightarrow \neg Error(PermsOf_g\ p_{init}\ tr')$$

3

Propiedades del Framework

En este capítulo se detallan los mecanismos para abstraer las políticas de concesión de permisos y se establecen criterios para su comparación y análisis. También se definen las políticas presentes en los modelos analizados y se prueban la relación existente entre las mismas.

3.1. Políticas de Concesión de Permisos

Dos variantes de políticas de concesión de permisos se analizan en [3]: dado un recurso de tipo rt, una de las políticas establece que cuando se otorga un nuevo permiso a este tipo de recurso, todos los permisos previos se sobreescriben. Esta política es llamada $grant_{ow}$. La otra política analizada establece que los permisos otorgados para rt se acumulan con los previamente obtenidos para ese tipo de recurso. Estas políticas se definen formalmente como sigue:

$$
\begin{aligned}
grant_{ow} :\ & grantPolicy \stackrel{def}{=} \\
& \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\
& update\ p\ rt\ pr\ m \\
grant_{ac} :\ & grantPolicy \stackrel{def}{=} \\
& \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\
& update\ p\ rt\ (pr \sqcup_{PermRes} (p\ rt).permRes)\ (m +_{mul} (p\ rt).mul)
\end{aligned}
$$

En [3] se analiza la expresividad del modelo y en particular, se afirma que los modos de permisos de MIDP pueden ser representados en el modelo. En este trabajo, adoptamos la representación propuesta por los autores. El término $grant_{bk}$ representa el modo de permisos blanket, que provee acceso irrestricto a un tipo de recursos. El modo de permisos one-shot, que provee un

solo acceso a un tipo de recurso dado se representa mediante el término $grant_{os}$.

$$\begin{aligned}
grant_{bk}: \quad & grantPolicy \stackrel{def}{=} \\
& \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\
& update\ p\ rt\ (pr \sqcup_{PermRes} (p\ rt).permRes)\ \infty \\
grant_{os}: \quad & grantPolicy \stackrel{def}{=} \\
& \lambda\ (p : Perm)\ (rt : ResType)\ (pr : PermRes\ rt)\ (m : Mul)\ . \\
& update\ p\ rt\ pr\ 1
\end{aligned}$$

El caso del modo de permisos allowed y session especificados en MIDP 2.0 pueden ser modelados con la política blanket. En el primer caso, los permisos concedidos se mantienen por el resto del ciclo de vida de la aplicación a la que se le otorga el permiso,mientras que en el segundo caso, el alcance de los permisos es la sesión en la que se otorga el permiso a la aplicación.

Con el objetivo de realizar un análisis comparativo de políticas de concesión de permisos como las que acabamos de definir, se define la siguiente relación:

$$\begin{aligned}
g_1 \sqsubseteq_g g_2 \stackrel{def}{=} \quad & \forall (tr : Trace)(p : Perm), \\
& Ptrace_{g_1}\ tr \rightarrow Safe_{g_1}\ p\ tr \rightarrow Safe_{g_2}\ p\ tr
\end{aligned}$$

Este orden establece que dado un grafo de control de flujo, para cualquier traza del grafo válida de acuerdo con g_1 y cualquier conjunto inicial de permisos, si la traza es segura cuando los permisos se conceden utilizando la política g_1 entonces también debe ser seguro si los permisos se conceden utilizando la política g_2. Intuitivamente, la política g_1 define un modelo de permisos más restrictivo que g_2.

El siguiente lema establece que la relación de orden entre estados de permisos preserva situaciones de error. También puede probarse que la relación $\sqsubseteq_g$ es un orden parcial (reflexivo, transitivo, antisimétrico). Estos resultados serán de ayuda cuando intentemos relacionar las políticas definidas.

Lemma 3.1.

$$Lemma\ lePermError : \forall (p_1\ p_2 : Perm), Error\ p_1 \rightarrow p_1 \sqsubseteq_{Perm} p_2 \rightarrow Error\ p_2$$

Los siguientes teoremas establecen condiciones suficientes para probar que dos políticas de concesión de permisos, g_1 y g_2 por ejemplo, están en la relación de orden ($g_1 \sqsubseteq_g g_2$):

1. las situaciones de error que surgen utilizando g_2 como política de concesión de permisos también surgen si se utiliza g_1,

2. si la política g_1 es aplicada, entonces todos los permisos disponibles al final de la traza también están disponibles si se utiliza g_2.

Este teorema resulta de mucha utilidad al tratar de comparar políticas de concesión de permisos.

Theorem 3.2.

$$
\begin{aligned}
&Theorem\ lePolicyCrit : \\
&\quad \forall (g_1\ g_2 : grantPolicy) \\
&\quad (H_{errors} : \forall (rt : ResType)\ (pr : validPermRes\ rt)\ (m : NZMulValid) \\
&\qquad\qquad (p : Perm), Error(g_2\ rt\ pr\ m\ p) \rightarrow Error(g_1\ rt\ pr\ m\ p)) \\
&\quad (H_{perms} : \forall (p : Perm)\ (tr : Trace), \\
&\qquad\qquad (PermsOf_{g_1}\ p\ tr) \sqsubseteq_{Perm} (PermsOf_{g_2}\ p\ tr)), \\
&\quad g_1 \sqsubseteq_g g_2
\end{aligned}
$$

Demostración. La prueba se realiza por inducción en $(PTrace_{g_1}\ tr)$, que se obtiene luego de reemplazar $g_1 \sqsubseteq_g g_2$ por su definición. Si la traza tr es vacía, entonces el teorema se cumple trivialmente. En el caso que la traza sea un singleton (de un solo elemento), la prueba utiliza la hipótesis H_{errors} y luego se realiza análisis por casos en el tipo de instrucción asociada con ese nodo, el caso interesante corresponde a la instrucción *Grant* (ya que las demás no afectan el estado de los permisos del dispositivo y *Consume* es monótona con respecto a $\sqsubseteq_{Perm}$)

El paso inductivo se prueba utilizando el lema *lePermError*, la hipótesis H_{perms} y la hipótesis de inducción. □

3.2. Relación entre Políticas de Concesión de Permisos

Utilizando el marco formal definido hasta ahora, es posible establecer y demostrar un teorema que establece como las cuatro políticas descriptas en la sección previa están relacionadas de acuerdo con la relación de orden $\sqsubseteq_g$.

Theorem 3.3.

$$Theorem\ grantPolicyRel : grant_{os} \sqsubseteq_g grant_{ow} \sqsubseteq_g grant_{ac} \sqsubseteq_g grant_{bk}$$

Demostración. La prueba de este teorema sigue de probar cada una de las tres relaciones $grant_{os} \sqsubseteq_g grant_{ow}$, $grant_{ow} \sqsubseteq_g grant_{ac}$ y $grant_{ac} \sqsubseteq_g grant_{bk}$, para luego aplicar transitividad del orden $\sqsubseteq_g$. Cada desigualdad es probada aplicando el teorema que establece las condiciones suficientes para probar que dos políticas de concesión de permisos están en la relación de orden (Teorema 3.2), y siguiendo una estrategia similar a la utilizada en su demostración. Aquí se presenta en detalle la prueba de la primera desigualdad y también se comenta brevemente como se obtienen los dos casos restantes.

La aplicación del Teorema 3.2 para probar $grant_{os} \sqsubseteq_g grant_{ow}$ requiere la prueba de las siguientes obligaciones de prueba:

1. $\forall (rt : ResType)\ (pr : validPermRes\ rt)\ (m : NZMulValid)\ (p : Perm),$
   $Error(g_2\ rt\ pr\ m\ p) \rightarrow Error(g_1\ rt\ pr\ m\ p))$
2. $\forall (p : Perm)\ (tr : Trace), (PermsOf_{g_1}\ p\ tr) \sqsubseteq_{Perm} (PermsOf_{g_2}\ p\ tr))$

La prueba de (1) se sigue de la aplicación del lema $lePermError$. Esto lleva a tener que probar que $(grant_{os}\ rt\ pr\ m\ p) \sqsubseteq_{Perm} (grant_{ow}\ rt\ pr\ m\ p)$. Al reemplazar $grant_{ow}$ y $grant_{os}$ por sus definiciones y aplicando los lemas que caracterizan a la función *update*, la obligación de prueba restante es $\langle pr, 1\rangle \sqsubseteq_{PermMul} \langle pr, m\rangle$ y $(p\ rt) \sqsubseteq_{PermMul} (p\ rt)$. La segunda es trivial dado que $\sqsubseteq_{PermMul}$ es un orden reflexivo. En lo que respecta a la primera, como $m : NZMulValid$ puede ser como mínimo 1, en cuyo caso, como $\sqsubseteq_{PermMul}$ es reflexivo, la obligación de prueba queda demostrada.

La prueba de (2) se realiza por inducción en tr:

- $tr = []$, la desigualdad se simplifica a $p \sqsubseteq_{Perm} p$ y dado que $\sqsubseteq_{Perm}$ es reflexivo, queda demostrada la obligación.
- $tr = tr' \triangleleft \langle n, st, ex\rangle$, la prueba se realiza utilizando análisis por casos en $KD\ n$. Los casos relevantes son *Grant* and *Consume* (dado que el resto de las instrucciones no afecta el estado de los permisos). El caso de la instrucción *Consume* es trivial, dado que la función *consume* es monótona, y por hipótesis de inducción sabemos que $(PermsOf_{grant_{os}}\ p\ tr') \sqsubseteq_{Perm} (PermsOf_{grant_{ow}}\ p\ tr')$. El caso correspondiente a la instrucción *Grant* se prueba utilizando la transitividad de $\sqsubseteq_{Perm}$, la hipótesis de inducción y los dos siguientes lemas:

- $\forall (rt : ResType)(pr : validPermRes\ rt)(m : NZMulValid)(p : Perm)$, $(grant_{os}\ rt\ pr\ m\ p) \sqsubseteq_{Perm} (grant_{ow}\ rt\ pr\ m\ p)$
- $\forall (rt : ResType)(pr : validPermRes\ rt)(m : NZMulValid)(p\ p' : Perm), p \sqsubseteq_{Perm} p' \to (grant_{ow}\ rt\ pr\ m\ p) \sqsubseteq_{Perm} (grant_{ow}\ rt\ pr\ m\ p')$.

Las pruebas de estos lemas se omiten por restricciones de espacio.

La estructura de las pruebas para las dos desigualdades restantes es bastante similar a la descripta previamente. En ambos casos la mayor parte de la prueba consiste en probar los lemas auxiliares similares a los que se utilizan para demostrar la obligación de prueba (2) para las políticas de concesión de permisos correspondientes. □

Este teorema y su prueba proveen evidencia formal de que, en primer lugar, de las cuatro políticas, MIDP one-shot es la más restrictiva y MIDP blanket es la más laxa. Además, estas dos políticas son relacionadas formalmente con las políticas de concesión de permisos definidas en [3] y éstas también son relacionadas, estableciendo que la política que acumula permisos es más permisiva que la que sobreescribe los permisos.

La diferencia entre acumular permisos y sobreescribirlos es sutil. Independientemente de cuál es más permisiva, ambas políticas poseen desventajas. En el caso de la política que acumula los permisos es que en cualquier punto del programa, para aproximar los permisos disponibles para un tipo de recurso dado, deben considerarse todas las concesiones y todos los consumos de permisos desde el comienzo del programa hasta el punto dado; mientras que en la política que acumula los permisos es suficiente con considerar la última concesión de permisos para ese tipo de recurso y todas los consumos subsiguientes. Esto sugiere que un análisis estático de permisos puede resultar más simple utilizando la política de sobre-escritura. Por otro lado, la política de sobre-escritura tiene un problema sutil. Es deseable que cuando uno otorga permisos para un recurso, los permisos resultantes sean al menos tan grandes como los que se tenía previamente, es decir, que la operación de concesión de permisos sea monótona con respecto a un orden entre los permisos. Este es el caso de la política que acumula, pero no de la que sobreescribe.

4

Generalización de Modelos de Permisos

En este capítulo describimos una generalización del framework presentado anteriormente. La misma se obtiene abstrayendo el reticulado subyacente al modelo de permisos mediante el sistema de módulos de Coq. También definimos un criterio de comparación entre modelos que permite caracterizar cuando un modelo es más general.

4.1. Abstracción de Modelos de Permisos

Los modelos de permisos estudiados en este trabajo se pueden caracterizar de manera general, notando que consisten de un reticulado subyacente, cuyos elementos representan permisos, junto con dos operaciones, grant y consume, que representan respectivamente como se conceden y como se consumen los permisos.

La noción de reticulado (Lattice) y sus propiedades ha sido modelada previamente en Coq por David Pichardie [13] y se utiliza en nuestra formalización. En ese trabajo, los reticulados son caracterizados mediante la siguiente signatura:

```
Module Type Lattice.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
```

Parameter $eq_trans : \forall\ x\ y\ z : t,\ eq\ x\ y \rightarrow eq\ y\ z \rightarrow eq\ x\ z.$

Parameter $eq_dec : \forall\ x\ y : t,\ \{eq\ x\ y\}+\{\neg\ eq\ x\ y\}.$

Parameter $order : t \rightarrow t \rightarrow$ Prop.

Parameter $order_refl : \forall\ x\ y : t,\ eq\ x\ y \rightarrow order\ x\ y.$

Parameter $order_antisym : \forall\ x\ y : t,\ order\ x\ y \rightarrow order\ y\ x \rightarrow eq\ x\ y.$

Parameter $order_trans : \forall\ x\ y\ z : t,\ order\ x\ y \rightarrow order\ y\ z \rightarrow order\ x\ z.$

Parameter $order_dec : \forall\ x\ y : t,\ \{order\ x\ y\}+\{\neg\ order\ x\ y\}.$

Parameter $join : t \rightarrow t \rightarrow t.$

Parameter $join_bound1 : \forall\ x\ y : t,\ order\ x\ (join\ x\ y).$

Parameter $join_bound2 : \forall\ x\ y : t,\ order\ y\ (join\ x\ y).$

Parameter $join_least_upper_bound : \forall\ x\ y\ z : t,\ order\ x\ z \rightarrow order\ y\ z \rightarrow order\ (join\ x\ y)\ z.$

Parameter $meet : t \rightarrow t \rightarrow t.$

Parameter $meet_bound1 : \forall\ x\ y : t,\ order\ (meet\ x\ y)\ x.$

Parameter $meet_bound2 : \forall\ x\ y : t,\ order\ (meet\ x\ y)\ y.$

Parameter $meet_greatest_lower_bound : \forall\ x\ y\ z : t,\ order\ z\ x \rightarrow order\ z\ y \rightarrow order\ z\ (meet\ x\ y).$

Parameter $bottom : t.$

Parameter $bottom_is_bottom : \forall\ x : t,\ order\ bottom\ x.$

End *Lattice.*

Esta signatura especifica que para ser un lattice, un módulo debe definir un conjunto t de valores, una noción de igualdad decidible sobre t que sea una relación de equivalencia, un orden decidible que sea reflexivo, transitivo y antisimétrico, dos operaciones binarias sobre elementos de t, *meet* y *join* que se comporten como greatest lower bound y least upper bound con respecto al orden y un elemento $\perp$ que sea el mínimo de t con respecto al orden.

Basándonos en esta signatura, nuestra noción de modelo de permisos extiende a la signatura de los lattices requiriendo la definición de dos funciones que representan la concesión y el consumo de permisos:

Module Type *Perm.*

Declare Module $L : Lattice.$

Parameter $consume : L.t \rightarrow L.t \rightarrow L.t.$

Parameter $cons_dec : \forall\ p\ q{:}\ L.t,\ L.order\ (consume\ q\ p)\ p.$

Parameter $cons_mon : \forall\ p\ q\ r : L.t,\ L.order\ p\ q \rightarrow$
$L.order\ (consume\ r\ p)\ (consume\ r\ q).$

Parameter $grant : L.t \rightarrow L.t \rightarrow L.t.$

```
    Parameter cons_inc : ∀ p q: L.t, L.order p (grant q p).
    Parameter grant_mon : ∀ p q r : L.t, L.order p q →
                          L.order (grant r p) (grant r q).
End Perm.
```

Estas funciones deben satisfacer ciertas propiedades, en particular, luego de una concesión de permisos debo obtener más permisos de los que tenía y luego de un consumo, menos. Notar que la política de concesión de permisos de sobre-escritura no satisface esta propiedad. Además, ambas operaciones deben ser monótonas.

A continuación definimos el modelo de permisos de MIDP como un módulo que satisface la signatura definida anteriormente:

```
Inductive midpPerm : Set :=
    | blanket : midpPerm
    | oneshot : midpPerm
    | none : midpPerm.
Definition eq_midp (p q: midpPerm) :=
    match (p, q) with
        | (blanket,blanket) ⇒ True
        | (oneshot,oneshot) ⇒ True
        | (none,none) ⇒ True
        | _ ⇒ False
    end.
Definition order_midp (p q: midpPerm) :=
    match (p, q) with
        | (_,blanket) ⇒ True
        | (oneshot,oneshot) ⇒ True
        | (none,_) ⇒ True
        | _ ⇒ False
    end.
```

Dadas las definiciones previas, construimos un módulo *Midp_lattice* de tipo Lattice. Para hacerlo, debemos dar todas las definiciones especificadas por la signatura Lattice:

```
Module Midp_lattice <: Lattice
    with Definition t := midpPerm
```

```
with Definition eq := eq_midp
with Definition order := order_midp
with Definition bottom := none.
Definition t := midpPerm.
Definition eq := eq_midp.
Definition order := order_midp.
Lemma eq_refl : ∀ x : t, eq x x.
Lemma eq_sym : ∀ x y : t, eq x y → eq y x.
Lemma eq_trans : ∀ x y z: t, eq x y → eq y z → eq x z.
Lemma eq_dec : ∀ x y, {eq x y} + {¬ eq x y}.
Lemma order_refl : ∀ x y : t, eq x y → order x y.
Lemma order_antisym : ∀ x y : t, order x y → order y x → eq x y.
Lemma order_trans : ∀ x y z : t, order x y → order y z → order x z.
Lemma order_dec : ∀ x y : t, {order x y}+{¬ order x y}.
Definition join : t → t → t.
intros t1 t2.
elim (order_dec t1 t2);intros;[exact t2 | exact t1].
Defined.
Lemma join_bound1 : ∀ x y : t, order x (join x y).
Lemma join_bound2 : ∀ x y : t, order y (join x y).
Lemma join_least_upper_bound : ∀ x y z : t, order x z → order y z → order (join x y) z.
Definition meet : t → t → t.
intros t1 t2.
elim (order_dec t1 t2);intros;[exact t1 | exact t2].
Defined.
Lemma meet_bound1 : ∀ x y : t, order (meet x y) x.
Lemma meet_bound2 : ∀ x y : t, order (meet x y) y.
Lemma meet_greatest_lower_bound : ∀ x y z : t, order z x → order z y → order z (meet x
y).
Definition bottom := none.
Lemma bottom_is_bottom : ∀ x : t, order bottom x.
End Midp_lattice.
```

Las pruebas de las propiedades han sido omitidas por razones de espacio, pero están demostradas en el desarrollo en Coq. En general son triviales y consisten en análisis por casos y

simplificación. El lector interesado es referido una vez más a la especificación completa del framework disponible online.

Una vez definido el reticulado subyacente, podemos definir un módulo de tipo *Perm* que caracterice el modelo de permisos de MIDP de la siguiente manera:

```
Module midp_Perm<: Perm with Module L := Midp_lattice.
  Module L := Midp_lattice.
  Definition consume (p q: L.t): L.t :=
  match q with
  | none ⇒ none
  | oneshot ⇒ none
  | blanket ⇒ blanket
  end.
  Lemma cons_dec : ∀ p q: L.t, L.order (consume q p) p.
  Lemma cons_mon : ∀ p q r : L.t, L.order p q →
                                   L.order (consume r p) (consume r q).
  Definition grant := L.join.
  Lemma cons_inc : ∀ p q: L.t, L.order p (grant q p).
  Lemma grant_mon : ∀ p q r : L.t, L.order p q →
                                   L.order (grant r p) (grant r q).
End midp_Perm.
```

Análogamente, podemos definir el modelo de permisos presentado en el trabajo de Besson et al. Primero definimos el reticulado subyacente, es decir, definimos un módulo con signatura *Lattice*:

```
Inductive fmacmdPerm : Set :=
  | bot : fmacmdPerm
  | mul : nat → fmacmdPerm
  | inf : fmacmdPerm.
  Definition eq_fmacmd(p q : fmacmdPerm) :=
   match (p,q) with
       | (bot,bot)⇒ True
       | (mul m, mul n) ⇒ m = n
       | (inf, inf) ⇒ True
```

```
      | _ ⇒ False
    end.
  Definition order_fmacmd(p q : fmacmdPerm) :=
    match (p,q) with
      | (_,inf) ⇒ True
      | (mul m,mul n ) ⇒ le m n
      | (bot, _) ⇒ True
      | _ ⇒ False
    end.
```

Utilizando las definiciones anteriores, procedemos de la siguiente manera:

```
Module fmacmd_lattice <: Lattice
  with Definition t := fmacmdPerm
  with Definition eq := eq_fmacmd
  with Definition order := order_fmacmd
  with Definition bottom := bot.
  Definition t := fmacmdPerm.
  Definition eq := eq_fmacmd.
  Definition order := order_fmacmd.
  Lemma eq_refl : ∀ x : t, eq x x.
  Lemma eq_sym : ∀ x y : t, eq x y → eq y x.
  Lemma eq_trans : ∀ x y z: t, eq x y → eq y z → eq x z.
  Lemma eq_dec : ∀ x y, {eq x y} + {¬ eq x y}.
  Lemma order_refl : ∀ x y : t, eq x y → order x y.
  Lemma order_antisym : ∀ x y : t, order x y → order y x → eq x y.
  Lemma order_trans : ∀ x y z : t, order x y → order y z → order x z.
  Lemma order_dec : ∀ x y : t, {order x y}+{¬ order x y}.
  Definition join (n : t) (m : t) :=
  match (n,m) with
     | (inf, _) ⇒ inf
     | (_,inf) ⇒ inf
     | (bot, x) ⇒ x
     | (x, bot) ⇒ x
     | (mul n, mul m) ⇒ mul (nat_lattice.join n m)
  end.
```

```
    Lemma join_bound1 : ∀ x y : t, order x (join x y).
    Definition join_bound2 : ∀ x y : t, order y (join x y).
    Definition join_least_upper_bound : ∀ x y z : t, order x z → order y z → order (join x y)
z.
    Definition meet (n : t) (m : t) :=
    match (n,m) with
        | (inf, x) ⇒ x
        | (x,inf) ⇒ x
        | (bot, _) ⇒ bot
        | (_, bot) ⇒ bot
        | (mul n, mul m) ⇒ mul (nat_lattice.meet n m)
    end.
    Lemma meet_bound1 : ∀ x y : t, order (meet x y) x.
    Lemma meet_bound2 : ∀ x y : t, order (meet x y) y.
    Lemma meet_greatest_lower_bound : ∀ x y z : t, order z x → order z y → order z (meet x
y).
    Definition bottom := bot.
    Lemma bottom_is_bottom : ∀ x : t, order bottom x.
End fmacmd_lattice.
```

Ahora utilizamos el módulo definido arriba para definir un módulo que implemente la estructura de permisos:

```
Module fmacmd_Perm <: Perm with Module L := fmacmd_lattice.
    Module L := fmacmd_lattice.
    Definition consume(_ m : L.t) : L.t:=
    match m with
        | bot ⇒ bot
        | (mul O) ⇒ bot
        | (mul (S O))=¿bot
        | (mul (S n)) ⇒ mul n
        | inf ⇒ inf
     end.
    Lemma cons_dec : ∀ p q: L.t, L.order (consume q p) p.
    Lemma cons_mon : ∀ p q r : L.t, L.order p q →
                                        L.order (consume r p) (consume r q).
```

```
Definition grant(m n: L.t) : L.t:=
match (m,n) with
  | (bot, n) ⇒ n
  | (inf, _) ⇒ inf
  | (n, bot) ⇒ n
  | (_, inf) ⇒ inf
  | (mul p, mul q) ⇒ mul (max p q)
 end.
Lemma cons_inc : ∀ p q: L.t, L.order p (grant q p).
Lemma grant_mon : ∀ p q r : L.t, L.order p q →
                                   L.order (grant r p) (grant r q).
End fmacmd_Perm.
```

Una vez definida la noción de modelo de permiso, podemos definir todas las construcciones de el cap. 2 con módulos parametrizados por Perm. La descripción de estos módulos se omite, ya que sus definiciones coinciden en muchos aspectos con aquellas dadas en dicho capítulo.

En particular, mediante esta generalización podemos obtener dos modelos de programas correspondientes al modelo de permisos de MIDP y al modelo de permisos de Besson et al. a partir de la aplicación de un functor:

```
Module midpPrograms := Prog midp_Perm.
Module fmacmdPrograms := Prog fmacmd_Perm.
```

y de manera análoga podemos obtener la noción de traza:

```
Module Trace_midp := Trace midp_Perm.
Module Trace_fmacmd := Trace fmacmd_Perm.
```

De esta forma hemos obtenido una generalización del modelo parametrizada completamente por la estructura de los permisos definidos y por su reticulado subyacente. Se obtienen todas las construcciones del capítulo 2 como un caso particular de esta generalización. De alguna forma, se puede pensar entonces que las construcciones de dicho capítulo se vuelven irrelevantes. Quizás ese sea el caso, pero también es cierto que sin dicho trabajo de formalización, difícilmente se hubiese alcanzado la generalización presentada aquí.

4.2. Noción de Generalidad de Modelos de Permisos

En esta sección nos proponemos obtener un concepto formal que caracterice la noción de cuando un modelo de permisos es más general que otro. Intuitivamente, ocurre esta situación

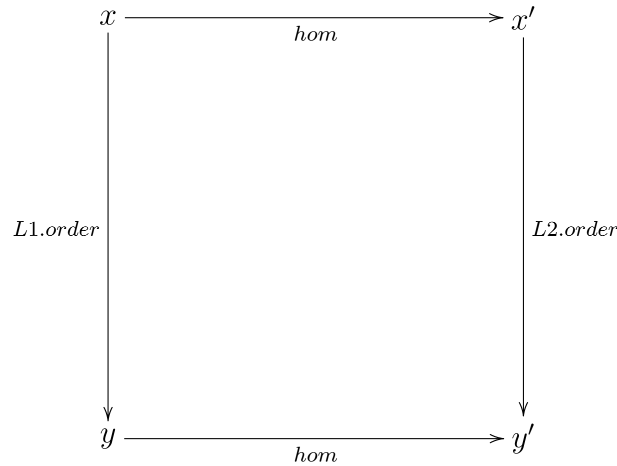

Figura 4.1: Diagrama Conmutativo que representa la Monotonía del Homomorfismo

cuando los permisos de uno de los modelos pueden ser mapeados a los permisos del otro, respetando la estructura, es decir, respetando el orden entre los permisos y las operaciones definidas sobre éstas. En la fig. 4.1 se detalla esta propiedad en forma de diagrama conmutativo. Este mapeo no es más que un homomorfismo entre los reticulados subyacentes a los modelos de permisos.

A continuación damos una caracterización mediante una signatura Coq de homomorfismos de reticulados:

Module Type *Lat_hom*.

Declare Module *L1* : *Lattice*.

Declare Module *L2* : *Lattice*.

Parameter *hom* : *L1.t* → *L2.t*.

Parameter *hom_mon* : ∀ *x1 x2* : *L1.t* ,
L1.order x1 x2 → *L2.order* (*hom x1*) (*hom x2*).

Parameter *hom_bot* : *hom L1.bottom* = *L2.bottom*.

Parameter *hom_join* : ∀ *x1 x2* : *L1.t*,
hom (*L1.join x1 x2*) = *L2.join* (*hom x1*) (*hom x2*).

Parameter *hom_meet* : ∀ *x1 x2* : *L1.t*,
hom (*L1.meet x1 x2*) = *L2.meet* (*hom x1*) (*hom x2*).

End *Lat_hom*.

La noción de homomorfismo entre reticulados puede ser extendida a homomorfismos entre modelos de permisos. Las idea es que el homomorfismo entre los reticulados subyacentes debe también preservar la estructura de los permisos. En la fig. 4.2 se detallan las propiedades

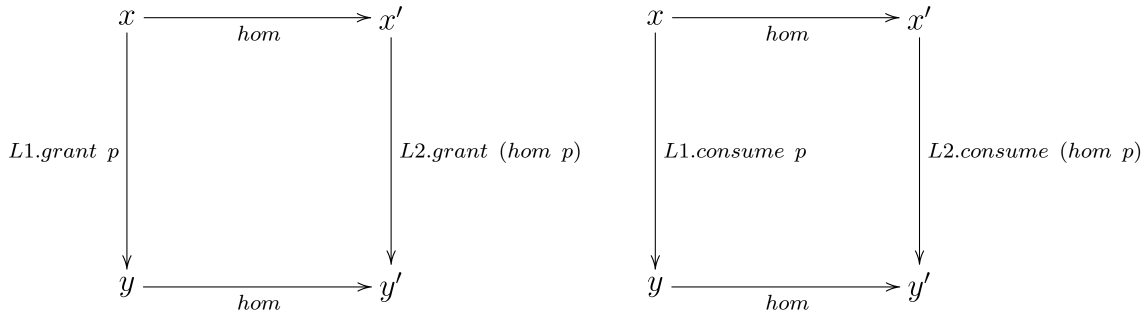

Figura 4.2: Propiedad de Homomorfismos entre Estructura de Permisos

que debe satisfacer el homomorfismo, con respecto a *grant* y *consume* en forma de diagrama conmutativo.

Esto se captura con la siguiente signatura Coq:

```
Module Type Perm_hom.
  Declare Module L1 : Lattice.
  Declare Module L2 : Lattice.
  Declare Module P1 : Perm
  with Module L := L1.
  Declare Module P2 : Perm
  with Module L := L2.
  Declare Module L_hom : Lat_hom
  with Module L1 := L1
  with Module L2 := L2.
  Parameter hom_grant : ∀ p q,
              L_hom.hom (P1.grant p q) = P2.grant (L_hom.hom p)(L_hom.hom q).
  Parameter hom_consume : ∀ p q,
              L_hom.hom (P1.consume p q) = P2.consume (L_hom.hom p)(L_hom.hom
q).
End Perm_hom.
```

En conclusión, hemos definido una caracterización formal de generalidad de estructura de permisos. La existencia de un homomorfismo entre reticulados subyacentes (con propiedades adicionales) define un orden de generalidad entre las estructuras de permisos. Es fácil ver que:

- El orden es reflexivo ya que la identidad es un homomorfismo entre estructuras de permisos

- El orden es transitivo ya que los homomorfismos son cerrados por composición
- El orden es antisimétrico si se considera igualdad “up to isomorphisms”, ya que dos homomorfismos en direcciones opuestas, se componen formando la identidad.

Por lo tanto, el orden definido representa un orden parcial (poset).

Una utilidad adicional de este homomorfismo subyacente es que puede extenderse para crear homomorfismos entre programas y trazas de los modelos relacionados por él. Al definir un functor con esta signatura:

Module Type *Apply* (*P* : *Perm*) (*H* : *Perm_Hom*) : *Perm*

Luego utilizando este functor, se puede, por ejemplo:

Module *Program_m1* := *Program (Apply m1_Perm m1_m2_hom)*

De esta forma, podemos obtener un modelo de programas de $m1$ representado por programas de $m2$. Un tratamiento análogo puede hacerse para trazas. Esto puede resultar de utilidad para comparar cuán restrictivos son modelos de seguridad con reticulados subyacentes diferentes, pero relacionados por un homomorfismo.

4.3. Homomorfismo entre los Modelos

A continuación, definimos un homomorfismo entre la estructura de permisos de MIDP y la de Besson et al., brindando evidencia formal de que el segundo es efectivamente más general que el primero. La idea intuitiva se puede ver en fig. 4.3.

Inicialmente, definimos el homomorfismo entre los reticulados subyacentes:

```
Module fm_midp_lat_hom <: Lat_hom
  with Module L1 := Midp_lattice
  with Module L2 := fmacmd_lattice.
  Module L1 := Midp_lattice.
  Module L2 := fmacmd_lattice.
  Definition hom : Midp_lattice.t → fmacmd_lattice.t :=
  fun p ⇒ match p with
  | none ⇒ bot
  | oneshot ⇒ mul 1
  | blanket ⇒ inf
  end.
  Lemma hom_mon : ∀ x1 x2 : L1.t ,
```

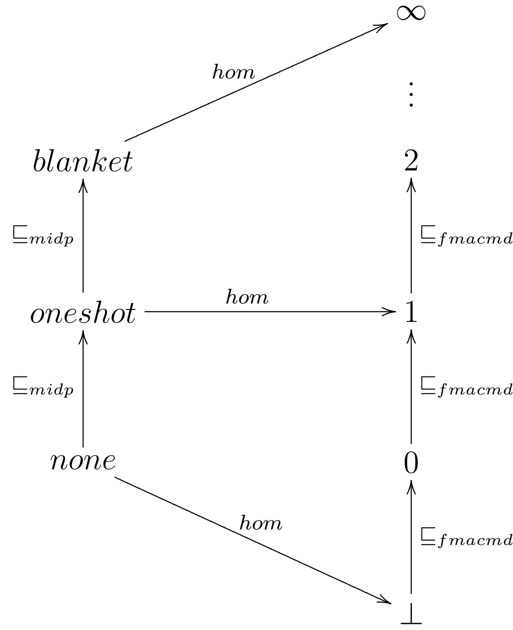

Figura 4.3: Diagrama que representa el Homomorfismo entre la Estructura de Permisos de MIDP y la del Modelo de [3]

$L1.order\ x1\ x2 \rightarrow L2.order\ (hom\ x1)\ (hom\ x2).$

Lemma $hom_bot : hom\ L1.bottom = L2.bottom.$

Lemma $hom_join : \forall\ x1\ x2 : L1.t,$

$hom\ (L1.join\ x1\ x2) = L2.join\ (hom\ x1)\ (hom\ x2).$

Lemma $hom_meet : \forall\ x1\ x2 : L1.t,$

$hom\ (L1.meet\ x1\ x2) = L2.meet\ (hom\ x1)\ (hom\ x2).$

End $fm_midp_lat_hom.$

Luego utilizamos esta definición para definir el homomorfismo entre estructuras de permisos:

Module $fm_midp_perm_hom : Perm_hom$

with Module $L1 := Midp_lattice$

with Module $L2 := fmacmd_lattice.$

Module $L1 := Midp_lattice.$

Module $L2 := fmacmd_lattice.$

Module $P1 := midp_Perm.$

Module $P2 := fmacmd_Perm.$

Module $L_hom := fm_midp_lat_hom.$

Lemma $hom_grant : \forall\ p\ q,$

$L_hom.hom\ (P1.grant\ p\ q) = P2.grant\ (L_hom.hom\ p)(L_hom.hom\ q).$

Lemma $hom_consume$: ∀ p q,

$L_hom.hom\ (P1.consume\ p\ q) = P2.consume\ (L_hom.hom\ p)(L_hom.hom\ q)$.

End $fm_midp_perm_hom$.

Nuevamente hemos omitido las pruebas por razones de espacio.

Este homomorfismo provee evidencia formal acerca de la mayor generalidad del modelo de seguridad propuesto por Besson et al. Así mismo, nos brinda la posibilidad de expresar uno en términos del otro de manera concisa y elegante.

5

Conclusiones y Trabajos Futuros

Esta tesina reporta trabajo relacionado con la especificación formal y el análisis de modelos de control de acceso para dispositivos móviles interactivos.

Se ha presentado un framework novedoso, formalizado utilizando el asistente de pruebas Coq, que provee un marco uniforme para definir y analizar modelos de control de acceso que incorpora mecanismos interactivos de concesión y consumo de permisos.

En particular, el trabajo presentado aquí, se enfocó en dos modelos de permisos distinguidos: el definido en la versión 2.0 de MIDP y el definido por Besson et al. en [3]. Una de las desventajas del modelo de permisos MIDP, es que el usuario es forzado a elegir entre tediosas y continuas interrupciones en aplicaciones interactivos para conceder permisos (one-shot) o confiar totalmente en las aplicaciones al otorgarles permisos casi irrestrictos para acceder a recursos sensibles. El modelo propuesto en [3] es más flexible que el de MIDP, permitiendo posibilidades adicionales en la forma en la que se conceden los permisos. Se ha definido una caracterización de ambos modelos en términos de una definición formal de política de concesión de permisos.

Otro tipo de políticas de permisos pueden ser expresadas también en este framework. En particular, puede ser adaptado para introducir una noción de revocación de permisos, un modo de permisos no considerado en MIDP. Una revocación de permisos puede ser modelada, en la política de sobre-escritura de permisos, por ejemplo, como una concesión con multiplicidad 0 a un tipo de recurso. En la política de acumulación de permisos podría ser modelada introduciendo multiplicidades negativas. La introducción del concepto de revocación de permisos permitiría, sin otros cambios, modelar una noción de “scope” de permisos. Ese “scope” puede ser considerado como el intervalo de la sesión delimitado por una activación y una revocación de ese tipo de permiso.

Una relación de orden $\sqsubseteq_g$ entre políticas de concesión de permisos fue también presentada

en el trabajo. Se establecieron dos teoremas y se discutieron sus demostraciones: uno que establece condiciones necesarias para probar que dos políticas de concesión de permisos están relacionadas por el orden, y otro que establece un criterio de comparación preciso y formal entre las políticas de concesión de permisos definidas por los modelos. En particular, se ha demostrado formalmente que la política acumulativa es más permisiva que la política de sobre-escritura. Adicionalmente se ha demostrado que $\sqsubseteq_g$ define un reticulado con las políticas $grant_{os}$ y $grant_{bk}$ como los elementos $\perp$ y $\top$ respectivamente, constituyendo un marco algebraico formal en el cual las políticas de concesión de permisos pueden ser relacionadas de manera precisa y comparadas.

En el capítulo 4 se ha definido una generalización del framework que abstrae completamente el reticulado subyacente del modelo de permisos. Esto es muy útil ya que el modelo de programas y trazas quedan parametrizados por el reticulado y la forma en la que se otorgan y consumen los permisos. Esto es de gran utilidad ya que con solo definir estas últimas construcciones, se puede obtener un modelo completo mediante la aplicación de un functor. Además se establece un criterio de comparación formal entre modelos de permisos, basado en la noción de homomorfismo. Se prueba que este criterio define un orden de generalidad entre las estructuras de permisos, que es reflexivo, transitivo y simétrico. También se define el homomorfismo entre el modelo de permisos de MIDP y el de Besson et al. en [3], testigo de que efectivamente el segundo es más general que el primero.

Trabajos futuros en esta línea son el estudio y la especificación, utilizando el marco formal definido por el framework, de algoritmos para forzar políticas de seguridad derivadas de distinto tipo de modelos de permisos para controlar el acceso a recursos sensibles de dispositivos. Uno de los principales objetivos a futuro es extender el framework para permitir la construcción de prototipos certificados a partir de las definiciones formales de esos algoritmos.

Bibliografía

[1] M. Bartoletti, P. Degano, and G-L. Ferrari. Static analysis for stack inspection. In *Electronic Notes in Computer Science*, volume 54, 2001.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[3] F. Besson, G. Dufay, and T. Jensen. A formal model of access control for mobile interactive devices. In *11th European Symposium on Research in Computer Security (ESORICS'06), LNCS 4189*, pages 110–126, 2006.

[4] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model ckecking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[5] S. Zanella Béguelin, G. Betarte, and C. Luna. A formal specification of the MIDP 2.0 security model. In *Formal Aspects in Security and Trust, Fourth International Workshop, FAST 2006, Hamilton, Ontario, Canada, August 26-27, 2006, Revised Selected Papers*, volume 4691 of *LNCS*, pages 220–234. Springer-Verlag, 2006.

[6] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[7] T. Coquand and G. Huet. The Calculus of Constructions. In *Information and Computation*, volume 76, pages 95–120. Academic Press, February/March 1988.

[8] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

[9] Juan Manuel Crespo, Gustavo Betarte, and Carlos Luna. A framework for the analysis of access control models for interactive mobile devices. In Stefano Berardi, Ferruccio Damiani, and Ugo de' Liguoro, editors, *TYPES 2008 Post-Proceedings*, Lecture Notes in Computer Science. Springer Computer Science Editorial, 2008.

[10] JSR 118 Expert Group. Mobile information device profile for java 2 micro edition. version 2.0. Technical report, Sun Microsystems, Inc. and Motorola, Inc., 2002.

[11] JSR 37 Expert Group. Mobile information device profile for java 2 micro edition. version 1.0. Technical report, Sun Microsystems, Inc., 2000.

[12] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *Proc. of the 20th IEEE Symp. on Security and Privacy*, pages 89–103. New York: IEEE Computer Society, may 1999.

[13] David Pichardie. Building certified static analysers by modular construction of well-founded lattices. In *Proc. of the 1st International Conference on Foundations of Informatics, Computing and Software (FICS'08)*, volume 212 of *Electronic Notes in Theoretical Computer Science*, pages 225–239, 2008.

[14] R. Roushani, G. Betarte, and C. Luna. A Certified Access Controller for JME-MIDP 2.0 enabled Mobile Devices. In *I Chilean Workshop on Formal Methods, Punta Arenas, Chile.* IEEE Computer Society, 2008. To appear.

[15] Sun Microsystems, Inc. *Java Platform Micro Edition.*, Last accessed: Oct. 2008. http://java.sun.com/javame/index.jsp.

[16] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, 2006.