Institut für Informatik, Lehrstuhl XII

Teams as Types

- A Formal Treatment of Authorisation in Groupware -

Wolfgang Naraschewski

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Johann Schlichter
Prüfer der Dissertation:	
	1. Univ.-Prof. Bernd Brügge, Ph.D.
	2. Prof. Dr. Dana Scott, Carnegie Mellon University Pittsburgh, Pennsylvania / USA

Die Dissertation wurde am 18.1.2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 7.3.2001 angenommen.

Acknowledgement

I would like to thank my supervisor Bernd Brügge for giving me the opportunity to work in this interesting project and for his visions which helped me guiding the way.

I also thank Dana Scott for being my referee and for getting involved into the topic of this thesis.

I am deeply indebted to Markus Wenzel who greatly influenced my work. In particular I would like to thank him for his patience and for his fruitful discussions.

I thank the – former and current – members of the Isabelle working group, namely Tobias Nipkow, David von Oheimb, Cornelia Pusch, Markus Wenzel and Leonor Prensa Nieto for their constructive collaboration and Manfred Broy for giving me the opportunity to work in this group.

This work would not have be possible without the patience of my wife. I am very grateful to her for giving me support. I also would like to thank Felix for giving me the creative moments.

Last but not least I would like to express my gratefulness to my parents Beate and Siegfried Naraschewski and to my parents in law Regina Hiller and Klaus Hiller for their continuous support of my work.

To my family Birgit, Felix and Kristina

Abstract

In the first part of the thesis we present a generic framework called Logos that may serve as a formal basis for investigations of authorisation issues in applications of groupware systems. Logos has been formalised within HOL – a well-known higher-order logic which is supported by a number of wide-spread theorem provers as e.g. Isabelle/HOL.

In the second part we present two applications of Logos. First we show, how it can be used to formalise the key authorisation concept of Lotus Notes – the groupware system currently dominating the market of commercial groupware solutions. Second we develop a case study (namely discussion bboards shared by a number of independent teams) demonstrating the usage of Logos as specification and verification environment for concrete safety critical applications.

Contents

Chapter 1

Introduction

"Entia non sunt multiplicanda praeter necessitatem"
(Wilhelm von Ockham 1285 - 1349)

Formal methods in computer science have benefited a lot from the security community. On the one hand, security issues have drawn attention to formal methods and on the other hand a lot of work has been funded by the security community. All formal methods that are developed to this end aim at developing correct and secure systems. Since 100% security can never be reached, we have to ask the question which approximations can be achieved. This question is too complex to be answered at hand and hence it is divided and conquered at lower levels. It is obvious that the problem can be best reduced to the different layers which distributed systems consist of. Ideally, the problem is defeated at the layers independently, but of course interactions can not be excluded.

Figures 1.1 and 1.2 depict the different layers of systems and their respective security guarantees. In [48] these layers and guarantees are discussed in more detail – we summarise the arguments in short words.

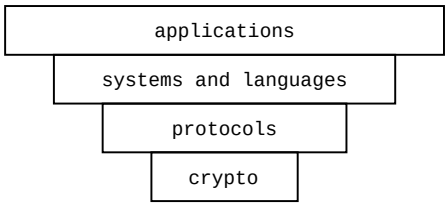


Figure 1.1: System layers

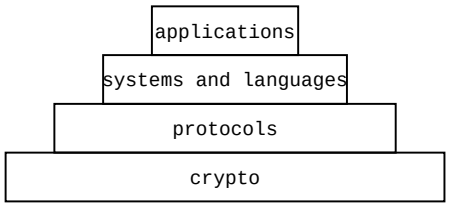


Figure 1.2: Security Guarantees

A solid cryptographic base, which provides means for encryption, decryption, signatures ..., is a precondition for any distributed, secure system. On top of this layer, protocols are used which serve the communication between the distributed entities. The systems and languages (built in the next layer)

use the services provided by the protocols and implement frameworks in which the applications (i.e. the programs that the users see) are realized.

Ironically, the degree of what we can guarantee is inversely proportional to the size of the layer (Figure 1.2). There are fundamental and deep results in cryptography. At the protocol level we have a handful of formal methods. For the systems and languages there is an increasingly but yet unsatisfactory degree of security. At the application layer we don't have much security guarantees at all. This thesis aims at tackling this gap for the special case of groupware (and in particular Lotus Notes) applications.

Of course, there is a good reason why the applicability of formal methods decreases as the applicability and size of the systems increases. Formal methods are only applicable to comparably small and clearly defined problems. Consequently, the major issue for formal investigations of system applications is to identify the core of the problem and to make quite a number of assumptions about the environment. Furthermore, the goal of completeness (which is a phantom anyway) has to be dropped.

But how to reduce the complexity? Figure 1.1 helps to answer this question systematically. The complexity can be reduced in the following three steps.

1. Focus only on the **layer of the application** taking all other layers as assumptions (environment).
2. Identify the **security critical core** of the application layer.
3. Find an adequate and **pragmatic mathematical model**.

Decisions taken at each of these steps have dramatic impact on the complexity and thus the applicability of the formal method. That's why we want to investigate the "design rationale" of the decisions we have taken in more detail.

Focus on application layer

There are two major security issues to be addressed by any groupware system:

- **authentication:** If a groupware system interacts with a subject who pretends to be "p" then authentication has to guarantee that the subject really is "p" and not someone else.
- **authorisation:** Authorisation assumes correct authentication. Depending on the subject's status the system decides if the subject may or may not perform a requested operation (e.g. read or write a document).

In actual groupware systems, authentication usually is guaranteed by public and private keys (as e.g. RSA) together with protocols used to control the interactions between the groupware system and the user. The details of authentication are dealt with in the layers "crypto" and "protocols" and hence can be ignored for the layer of the applications. Correct authentication is taken as axiom for the further development.

As far as authorisation is concerned, its realization is spread over the layers "systems and languages" and "applications".

Static vs. dynamic authorisation There are two ways of dealing with authorisation, which we call *static* and *dynamic*. Authorisation is *static* if it is determined globally for a larger part of an application and it only changes rarely over time. Usually, the access control is provided by the system itself and an application's manager only needs to apply the control mechanisms. For example, the database manager statically determines in the access control list of a Lotus Notes database which subjects may read documents. This assignment usually concerns all documents and remains unchanged throughout the life-time of the application. In contrast, authorisation is *dynamic* if access rights are context and time dependent and the implementation has to be provided explicitly for the application. For example, in on-line registrations for events, the right to submit a registration expires once the deadline has passed. To realize this requirement, a formula has to be implemented that is evaluated at runtime by the system.

As far as static authorisation is concerned, the groupware system is responsible for its correctness. This correctness can be checked once for the whole system and consequently holds for all of its applications. For dynamic authorisation the situation is different. The responsibility is imposed on the implementor. Correctness with respect to some specification has to be checked individually for each application. In security critical applications (which are standard in groupware) dynamic authorisation is a major stumbling block in providing security.

Dynamic authorisation: examples In the following we sketch two real world examples that illustrate the concept of dynamic authorisation (see Section 5.2).

The first example deals with a web-based intranet collectively shared by a set of scholarship foundations each of them offering a specific net for their own students and alumni. Each student may decide on a document basis which information is shared with other foundations.

The second example deals with a series of trans-atlantic academic student-projects involving students from Carnegie Mellon University and the Technische Universität München. All student projects are performed in cooperation with industrial partners. The students need to access

proprietary information that is subject to non-disclosure agreements. The problem is that part of the information needs to be shared between different project phases or projects with new team members whereas other confidential information needs to be kept secret. The author of each document determines if it needs to be treated according to the non-disclosure agreement. As the non-disclosure agreements impose hard security restrictions on the supporting groupware application, its correctness needs some certification. In this case, we have to ensure that confidential information may only be accessed by students who have signed a non-disclosure agreement.

Dynamic authorisation cannot be investigated isolated from the surrounding groupware system. Hence any formal method dealing with groupware applications also has to deal with the standard authorisation concept of the underlying groupware system.

Security critical core of the application

At the level of applications only authorisation issues are relevant for the security model. It is possible (and reasonable) to abstract away from all other security aspects which are hidden in the layers below.

As we have mentioned already, authorisation issues at the level of applications involve programs that are used to compute the access rights. Each groupware system supports its own language that is used to this end. Trying to imitate this language in the formal model would unnecessarily blow up the effort. From a pragmatically point of view it is much more reasonable to use an abstract programming or even only specification language and to leave the correspondence between the abstract model and the concrete programming language as a separate (not necessarily formally treated) problem.

Pragmatic mathematical model

Models for authorisation in groupware can – just as any other formal model – be achieved differently (the quotation goes back to C.A.R. Hoare):

> "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other ways is to make it so complicated that there are no obvious deficiencies"

This quotation describes two quite opposite approaches to formal models: deep or shallow embeddings. In a deep embedding syntax and semantics are formulised explicitly as inductive sets. Deep embeddings are suitable for investigations of the meta-theory of the framework itself, but ponderous

for its applications. Since our focus lies on applications, we have decided to ground on a shallow embedding, which reuses the syntax and semantics of the underlying logic by definitional extensions. In contrast with deep embeddings (e.g. see [1]), shallow embeddings inherit well-formedness properties from the underlying logic for free. By this decision we have taken Wilhelm von Ockham's razor quite seriously.

Framework Logos

"Logos" is a Greek term meaning "word". In history it has been used as a metaphor for human communication in general. The word "logic" has been derived from "logos", as "logic" originally had been concerned with formal argumentations. Groupware, which nowadays deals with electronic communication, is no longer connected with formal logic.

Re-connecting these two orthogonal topics requires bridging a wide gap between theory and practice. Each of the two communities of researchers has developed its own ideas and culture. The following example demonstrates how different contextual backgrounds hinder communication: In theory, type constructors are well-known. Type constructor "memory" for example constructs memories with data of a given, arbitrary type. A "group memory" then is a particular memory that stores groups. In contrast, "group memory" is often used in groupware to summarise all the knowledge a group has. In this thesis we try to establish a connection between these two orthogonal scientific lines of research (as indicated by the title "Teams as Types"). We propose a logical framework called "Logos" that may serve as a formal basis to investigate security (to be more precise authorisation) issues in groupware applications. Particularities of individual systems as Lotus Notes or BSCW then may be expressed within this framework (see Chapter 4).

The central question to be answered for Logos is which logical basis to take for the model. Since we want to develop a generic framework, the logical basis has to be flexible. We have decided upon HOL – a higher order logic. HOL can be understood as a version of typed set theory, with two distinct kinds of objects: terms denoting set theoretic individuals (numbers, tuples, functions etc.) and types denoting corresponding sets classifying the individuals. We have chosen HOL as it is well-known in theoretical computer science and it is supported by a number of wide-spread theorem provers as e.g. Isabelle/HOL. This allows for rigorous and machine checked specifications and correctness proofs of groupware authorisation issues.

Road map

In the next chapter (**Chapter 2**) we develop two groupware-independent theories as foundations for the subsequent sections. One is a theory of selected object-oriented concepts (see Section 2.2) which is the basis for Section 3.2.8 and the other is a theory of references (see Section 2.3) which we use throughout the paper. In **Chapter 3** we introduce the framework Logos which serves to express and validate authorisation issues in groupware systems and applications. The remaining chapters provide case studies which stress the applicability of Logos to concrete groupware systems and applications. **Chapter 4** refines the definitions of Logos to account for the core authorisation model of Lotus Notes – the leading system in the groupware market. The next chapter (**Chapter 5**) shows how concrete groupware applications are modelled in Logos. To this end a case study of shared discussion bboards is developed and a sample security property is proved formally. We conclude the thesis with some remarks on further and related work in **Chapter 6**.

Chapter 2

Foundations

In order to define the framework Logos for authorisation in groupware and to apply it to Lotus Notes and shared bboards the following groupware-independent theories are required. One is a theory of selected object-oriented concepts (see Section 2.2) which is the basis for Section 3.2.8 and the other is a theory of references (see Section 2.3) which we use throughout the paper. To stress that the theories are general and thus independent from our particular environment the running example for the section dealing with object-oriented concepts is the standard example from object-oriented literature (points) rather than a standard example from groupware.

Most mathematical papers take an informal understanding of the underlying logic as basis without making the employed logic explicit. This way reasoning gets simpler, frequently using so called "hand-waving" arguments. We wish to stress that this approach does not at all imply that the results do not withstand a rigorous formal investigation. Nevertheless, this approach eludes the chance to check the proofs with a mechanical proof-checker without extensive additional effort which is a great advantage in environments close to programming. In such environments the theorems usually are not profound but the proofs are quite extensive which often leads to careless mistakes. In this paper we make a compromise. Our theories have not been fully implemented in a theorem prover – thus omitting painful odds and ends. Nevertheless, we always have a particular formal logic (that is HOL – see [4]) in mind guaranteeing that all of our definitions and proofs could (with some straightforward effort) be certified in a theorem prover (that is Isabelle/HOL – see e.g. [27, 47, 31, 28]). The underlying logic HOL has also been implemented in several other theorem provers (see e.g. [9, 11]).

2.1 The HOL logic

In the following we sketch the underlying logic HOL. This short introduction is also published in [26].

2.1.1 Syntax and semantics

The syntax of HOL is that of simply-typed λ-calculus with a first-order language of types. *Types* are either variables α, or applications $(\tau_1, \ldots, \tau_n)\, t$; we drop the parentheses for $n \in \{0, 1\}$. Binary constructors are often written infix, e. g. function types $\tau_1 \to \tau_2$ (associating right). There is no way to bind type variables or make types depend on terms in HOL.

Terms are either typed constants c_τ or variables x_τ, applications $t\ u$ or abstractions[1] $\lambda x.\ t$. As usual, application associates to the left and binds most tightly. An abstraction body ranges from the dot as far to the right as possible. Nested abstractions like $\lambda x.\ \lambda y.\ t$ are abbreviated to $\lambda x\ y.\ t$. Terms have to be well-typed according to a standard set of typing rules.

HOL can be understood as a very simple version of typed set theory, with two distinct kinds of objects: terms denoting set theoretic individuals (numbers, tuples, functions etc.) and types denoting corresponding sets classifying the individuals. In ordinary untyped set theory everything is just a set, of course. We will often use lists rather than sets since lists are finite by definition. In finite cases, we will deliberately ignore their differences and use them interchangeably, though.

2.1.2 Theories

HOL theories consist of a *signature* part (declaring type constructors $(\alpha_1, \ldots, \alpha_n)\ t$ and polymorphic constant schemes $c : \sigma$) and *axioms*. All theories are assumed to contain a certain basis, including at least types *bool* and $\alpha \to \beta$ and several constants like logical connectives $\wedge, \vee, \Rightarrow : \mathit{bool} \to \mathit{bool} \to \mathit{bool}$, quantifiers $\forall, \exists : (\alpha \to \mathit{bool}) \to \mathit{bool}$ and equality $= : \alpha \to \alpha \to \mathit{bool}$.

Any theory induces a set of derivable theorems, depending on a fixed set of deduction rules that state several "obvious" facts of classical set theory.

Arbitrary axiomatisations are considered anathema in the HOL context. It is customary to use only *definitional extensions* (guaranteeing certain nice deductive and semantic properties) and honestly toil in deriving the desired properties from the definitions. HOL offers definition schemes for constants and types [35].

2.1.3 Constant definitions

The basic mechanism only admits introducing some axiom $\vdash c = t$ for a new constant c not occurring in t (and some further technical restrictions). We generalise the pure scheme to admit arguments of function definitions

[1] Binder λ is well-known in theoretical computer science. It allows for function definitions without having to introduce explicit names for the functions. The common definition $f(x) = t$ can be written equivalently $f = \lambda x.\ t$.

applied on the l.h.s. rather than abstracted on the r.h.s.: $\vdash f\,x\,y = t$ instead of $\vdash f = \lambda x\ y.\ t$. Furthermore, tuple abstraction, definitions by cases etc. may be written using ML-style pattern matching, e.g. $\vdash f\ (x, y) = t$ (which applies the pair eliminator $split : \alpha \times \beta \to (\alpha \to \beta \to \gamma) \to \gamma$ behind the scenes).

Later we will also use a proper extension of the HOL constant definition scheme, namely *overloading* [46]. Currently only the theorem prover Isabelle/HOL implements this. Here is a sample overloaded definition of some polymorphic constant $0 : \alpha$:

$$\begin{aligned} 0_{nat} &= zero \\ 0_{\alpha\times\beta} &= (0_\alpha, 0_\beta) \\ 0_{\alpha\to\beta} &= \lambda\, x_\alpha.\, 0_\beta \end{aligned}$$

Note that we do not have to cover all types of 0 here; additional clauses may be added later, provided overall consistency of the set of equations is preserved.

2.1.4 Type definitions

New polymorphic type schemes may be introduced in HOL systematically as follows: exhibiting a non-empty representing subset A of an existing type (with further technical restrictions) one may introduce a new axiom stating that $(\alpha_1, \ldots, \alpha_n)\ t$, for a new type constructor t, is in bijection with A. This basically identifies the new type with the representing subset.

HOL type definitions are peculiar as they only state equivalence up to isomorphism. There is no way to enforce actual equality, as do type conversions in type theories. As a consequence, the HOL algebra of types can be considered as freely generated (without loss of generality), always admitting an initial model where types of different names denote different sets. This freeness property will be quite important later for distinctness of record types. Even more fundamental, it underlies overloading [46], which is used in Section 2.2.2 to implement methods.

Paradoxically, more powerful logical systems like full set theory or the HOL-version underlying PVS (see [30]) are not quite suitable for our way of encoding methods, mainly because they no longer admit the freeness assumption of types.

2.2 Object-oriented concepts in HOL

Although HOL is a quite weak type theory, it is sufficiently expressive to model object-oriented concepts like inheritance or late-binding. These results have been published in [26] and are summarised in the following.

The encoding that is used is merely based on extensible records which we shall introduce now.

2.2.1 Extensible records

Extensible records are not new in literature (e.g. see [10]). Though we use a shallow encoding for Logos we may not use any calculus-based model as the one mentioned. We rather present a shallow encoding i.e. a merely definitional extension of our logic. Clearly, such an encoding has its drawbacks as far as expressivity is concerned, but from a pragmatic point of view it is much more applicable.

What are extensible records anyway?

Tuples and records Records are a minor generalisation of tuples, where components may be addressed by arbitrary labels (strings, identifiers, etc.) instead of just position. Our concrete record syntax is borrowed from ML (see [32] for a textbook): e.g. $\{x = a, y = b, z = c\}$ denotes an individual record of labels x, y, z and values a, b, c, respectively. The corresponding record type would be of the form $\{x : A, y : B, z : C\}$. Note that the labels contribute to record identity, consequently $\{x = 3, y = 5\}$ is completely different from $\{\mathit{foo} = 3, \mathit{bar} = 5\}$.

Just as records are no more than labelled products, there are also labelled sums: co-records. A co-record of type $\{\!|\ x : A, y : B\ |\!\}$ is either a value of type A or a value of type B. Each co-record comes with a set of constructors (one for each field). Our sample co-record would have constructors $x : A \to \{\!|\ x : A, y : B\ |\!\}$ and $y : B \to \{\!|\ x : A, y : B\ |\!\}$.

Record schemes Unlike ordinary tuples, records are better suited to a property oriented view in the sense of "record r has field l". As a concise means to refer to classes of records featuring certain fields we introduce *schemes*, both on the level of records and record types. Patterns of the form $\{x = a, y = b, \ldots\}$ refer to any record having at least fields x, y of value a, b, respectively. The corresponding type scheme is written as $\{x : A, y : B, \ldots\}$. The dots "..." are actually part of our notation and are pronounced "more". The more part of record schemes may be instantiated by zero or more further components. In particular, the concrete record $\{x = a, y = b\}$ is considered a (trivial) instance of the scheme $\{x = a, y = b, \ldots\}$.

As an example of relating records consider schemes $\{x = a, y = b, \ldots\}$ and $\{x = a, y = b, z = c, \ldots\}$. These are related in the sense that the latter is an extension of the former by addition of field $z = c$. On the level of types, one might say that any $\{x : A, y : B, z : C, \ldots\}$ is a *structural subtype* of $\{x : A, y : B, \ldots\}$. Note that (in our framework) record subtyping may only hold if the parent is an *extensible* record scheme. As a counterexample,

instances of $\{x : A, y : B, z : C, \ldots\}$ are *not* considered extensions of the *concrete* record type $\{x : A, y : B\}$.

With record schemes at the term and type level we have already "extensible records" at our disposal. In particular, we can define functions that operate on whole classes of records schematically, like $f\,\{x = a, y = b, \ldots\} = t$. Here the l.h.s. is supposed to bind variables a, b and "..." by pattern matching. To improve readability, we occasionally abbreviate $\{x = x, y = y, \ldots\}$ by $\{x, y, \ldots\}$, even on the r.h.s. provided this does not cause any ambiguity.

Before discussing encodings of this general concept of extensible records in formal logical systems we demonstrate its use by an example.

Example: abstract algebraic structures

Consider some bits of group theory: A *monoid* is a structure with carrier α and operations $\circ : \alpha \to \alpha \to \alpha$ and $1 : \alpha$ such that $\circ$ is associative and 1 is a left and right unit element (w.r.t. $\circ$). A *group* is a monoid with additional operation $inv : \alpha \to \alpha$ such that inv is left inverse (w.r.t. $\circ$ and 1). An *agroup* (abelian group) is a group where $\circ$ is commutative.

A well-known approach to abstract theories in HOL [9] uses n-ary predicates over the structures' operations (carrier types are included implicitly via polymorphism). Then *monoid* would be a predicate on pairs and *group*, *agroup* on triples as follows (below we use fancy syntax $\circ$, 1 for variables):

$$
\begin{array}{l}
monoid : (\alpha \to \alpha \to \alpha) \times \alpha \to bool \\
monoid\ (\circ, 1) = \forall x\ y\ z.\ (x \circ y) \circ z = x \circ (y \circ z)\ \wedge\ 1 \circ x = x\ \wedge\ x \circ 1 = x \\
group : (\alpha \to \alpha \to \alpha) \times \alpha \times (\alpha \to \alpha) \to bool \\
group\ (\circ, 1, inv) = monoid\ (\circ, 1)\ \wedge\ \forall x.\ (inv\ x) \circ x = 1 \\
agroup : (\alpha \to \alpha \to \alpha) \times \alpha \times (\alpha \to \alpha) \to bool \\
agroup\ (\circ, 1, inv) = group\ (\circ, 1, inv)\ \wedge\ \forall x\ y.\ x \circ y = y \circ x
\end{array}
$$

Note that *monoid* and *group*, acting on different signatures, do not admit an immediate notion of inclusion. To express that any group is a monoid one has to apply an appropriate forgetful functor first, mapping $(\circ, 1, inv)$ to $(\circ, 1)$.

We now use extensible records instead of fixed tuples to model algebraic structures. This will eliminate above problem of incompatible signatures, as record subtyping automatically takes care of this. Monoids are defined as follows:

$$
\begin{array}{l}
\textbf{record}\ \alpha\ \mathit{monoid\text{-}sig} = \\
\quad \circ : \alpha \to \alpha \to \alpha \qquad (\textbf{infix}) \\
\quad 1 : \alpha \\
\quad monoid : \{\circ : \alpha \to \alpha \to \alpha,\ 1 : \alpha,\ \ldots\} \to bool \\
\quad monoid\ \{\circ,\ 1,\ \ldots\} = \\
\qquad \forall x\ y\ z.\ (x \circ y) \circ z = x \circ (y \circ z)\ \wedge\ 1 \circ x = x\ \wedge\ x \circ 1 = x
\end{array}
$$

The record declaration introduces type scheme $\{\circ : \alpha \to \alpha \to \alpha,\ 1 : \alpha,\ \ldots\}$ together with several basic operations like constructors, selectors and updates (with the usual properties). Selectors are functions of the same name as the corresponding fields, e.g. $1 : \{\circ : \alpha \to \alpha \to \alpha,\ 1 : \alpha,\ \ldots\} \to \alpha$. Thus $(1\ \ M)$, which we often also write as $M.1$, refers to the unit element of structure M. The update operation for any field x is called *update-x*.

Based on this abstract theory of monoids, we may now introduce derived notions and prove generic theorems. For example, consider the following definition of exponentiation (by primitive recursion), together with an obvious lemma stating that $x^{m+n} = x^m \circ x^n$ holds in monoids:

$$
\begin{array}{l}
pow : \{\circ : \alpha \to \alpha \to \alpha,\ 1 : \alpha,\ \ldots\} \to nat \to \alpha \to \alpha \\
pow\ \ \{\circ,\ 1,\ \ldots\}\ \ 0\ \ x = 1 \\
pow\ \ \{\circ,\ 1,\ \ldots\}\ \ (Suc\ \ n)\ \ x = x \circ (pow\ \ \{\circ,\ 1,\ \ldots\}\ \ n\ \ x) \\
monoid\ \ M \Rightarrow M.pow\ \ (m + n)\ \ x = (M.pow\ \ m\ \ x)\, M.\circ\ \ (M.pow\ \ n\ \ x)
\end{array}
$$

Next we define groups as an extension of monoids as follows:

$$
\begin{array}{l}
\textbf{record}\ \alpha\ \ group\text{-}sig = \alpha\ \ monoid\text{-}sig + \\
\quad inv : \alpha \to \alpha \\
\quad group, agroup : \{\circ : \alpha \to \alpha \to \alpha,\ 1 : \alpha,\ inv : \alpha \to \alpha,\ \ldots\} \to bool \\
\quad group\ \ \{\circ,\ 1,\ inv,\ \ldots\} = monoid\ \ \{\circ,\ 1,\ inv,\ \ldots\} \wedge \forall x.\, (inv\ \ x) \circ x = 1 \\
\quad agroup\ \ \{\circ,\ 1,\ inv,\ \ldots\} = group\ \ \{\circ,\ 1,\ inv,\ \ldots\} \wedge \forall x\ y.\, x \circ y = y \circ x
\end{array}
$$

The *group-sig* type scheme has been defined as child of *monoid-sig* and directly inherits all primitive and derived operations (in particular selectors etc.). Apparently, any $\{\circ,\ 1,\ inv,\ \ldots\}$ is also an instance of $\{\circ,\ 1,\ \ldots\}$. Therefore, functions operating on the latter, also work on the former. For example consider the instance $pow\ \ \{\circ,\ 1,\ inv,\ \ldots\}$ for exponentiation on group structures.

By using extensible records we got for free what had to be done by explicit coercions (type casts) in other systems. Even more: apart from adapting *argument* types, *result* types are instantiated as well in our setting. As an example consider the following "functor" that reverses the binary operation of monoids:

$$rev\ \ \{\circ,\ 1,\ \ldots\} = \{\circ = \lambda x\ y.\, y \circ x,\ 1 = 1,\ \ldots\}$$

This function generically maps objects of type *monoid-sig* to *monoid-sig* and *group-sig* to *group-sig*:

$$
\begin{array}{l}
rev : \{\circ : \alpha \to \alpha \to \alpha,\ 1 : \alpha,\ \ldots\} \to \{\circ : \alpha \to \alpha \to \alpha,\ 1 : \alpha,\ \ldots\} \\
rev : \{\circ : \alpha \to \alpha \to \alpha,\ 1 : \alpha,\ inv : \alpha \to \alpha,\ \ldots\} \\
\qquad \to \{\circ : \alpha \to \alpha \to \alpha,\ 1 : \alpha,\ inv : \alpha \to \alpha,\ \ldots\}
\end{array}
$$

Note that a naive approach with type casts would have yielded only *group-sig* to *monoid-sig* in the latter case.

In our setting, the type system will always take care of adapting the signatures of the mathematical structures automatically. Actual structures are restricted by additional logical properties, though, as expressed by the predicates *monoid*, *group*, *agroup*. Using simple properties of monoids and groups, like $x \circ (inv\ x) = (inv\ x) \circ x$, we may actually prove that all three kinds of structures are logically invariant under the *rev* functor:

$$\begin{array}{l} monoid\ M \Rightarrow monoid\ M.rev \\ group\ G \Rightarrow group\ G.rev \\ agroup\ G \Rightarrow agroup\ G.rev \end{array}$$

In general, functors may not propagate that nicely down the hierarchy of algebras. If so, one might want to consider changing the meaning of such operations depending on the actual type of the argument structure. For example, some functor on monoids might be redefined on groups in order to take the additional *inv* field into account. Redefining functions this way amounts to *overriding methods* in object-oriented parlance (see Section 2.2.2 of how to achieve this).

Basic usage

A representation in untyped set theory Thinking in ordinary mathematics one may model extensible records as follows [16, Section 2.7.2]: fixing a set L of labels and a family of sets of values $(A_l)_{l \in L}$, the set of extensible records over these shall be the (dependent) partial function space $l \in L \rightharpoonup A_l$. That is, any record r is a partial function such that $r(l) \in A_l$, if $r(l)$ is defined. For example, record $\{x = 3, y = 5\}$ would be the function $r\colon x \mapsto 3, y \mapsto 5$, undefined elsewhere.

This encoding is rather "deep", labels and values are both first class individuals. We can express many notations of extensible records directly within the system as set theoretic functions or predicates. In particular, the relation "r has component l" would be "$r(l)$ is defined". Furthermore, relation "r' extends r" and operations "add component $l = x$ to r", "merge r and r'" could be expressed via set inclusion, insertion, union, respectively. Also note that these records are commutative: records $\{x = 3, y = 5\}$ and $\{y = 5, x = 3\}$ are equal.

Encoding

A deep encoding in HOL? Above encoding of records would in principle also work in HOL. We could encode partial functions as relations, or total functions to a range type with explicit *undefined* element. There is a snag, though, making this version of records very awkward to use in practice: it doesn't fit very well within the HOL type system. In particular, the sets of values A_l from above would have to be within the same type! If one

wanted to have different HOL types for different fields, explicit injections were required (via disjoint sums).

A better encoding of records in HOL should try to exploit the type system as much as possible. Such a representation would be much preferable even if it lost some of the properties and expressiveness of the set theoretic version. This is yet another example of applied logic within a concrete working environment where pure expressiveness may be quite unrelated to usefulness.

Shallow encoding of records in HOL To make a long story short, extensible records are just tuples that contain an extra "more" variable for possible extensions. Ignoring the fact that field names contribute to record identity for a while, the representation of $\{x = 3, y = 5, f = \mathit{true}, \ldots\}$ is just $(3, (5, (\mathit{true}, \mathit{more})))$ where *more* is a suitable term variable. The corresponding type $\{x : \mathit{int}, y : \mathit{int}, f : \mathit{bool}, \ldots\}$ is a nested product $(\mathit{int} \times (\mathit{int} \times (\mathit{bool} \times \alpha)))$, for some free type variable α.

Refining the *more* slot yields instances with additional fields, for example $\{x = 3, y = 5, f = \mathit{true}, z = 42, \ldots\}$ represented by the tuple $(3, (5, (\mathit{true}, (42, \mathit{more}'))))$. Containing free variables, record schemes are not basic values. Typically, they only appear in definitions of generic functions where *more* is bound by functional abstraction. On the level of types, the *more* position amounts to polymorphism.

Actual concrete record values can be achieved by instantiating the more slot to $()$, the sole element of the *unit* type, thus terminating the chain of record fields properly without affecting the semantics. For example, record $\{x = 3, y = 5, f = \mathit{true}\}$ would be $(3, (5, (\mathit{true}, ())))$, and consequently its type $\{x : \mathit{int}, y : \mathit{int}, f : \mathit{bool}\}$ would be $(\mathit{int} \times (\mathit{int} \times (\mathit{bool} \times \mathit{unit})))$.

We now focus again on labels. These shall act as a means to distinguish records with different field names. As we have already said earlier, HOL's algebra of types is so weak that it admits a freeness assumption: types of different names can never be enforced to be actually the same within the logic. This gives rise to the following technique to make field names contribute to record identity without having to bother about labels as first-class individuals.

For any field $x : \sigma$ we introduce an isomorphic copy of the HOL pair type $\times$ by type definition, calling it $\times_x$. We also obtain copies of the pair constructor and projections etc., with their usual properties. The copied constructor shall be $\mathit{x\text{-}field} : \sigma \to \beta \to \sigma \times_x \beta$. It is declared only at an instance of the general scheme $\alpha \to \beta \to \alpha \times_x \beta$ in order to obey the type constraint for field x as specified in the record type declaration.

Using a separate pair type for any field we now get the following shallow encoding of records: record $\{x = 3, y = 5, f = \mathit{true}, \ldots\}$ is

(*x-field* 3 (*y-field* 5 (*f-field true more*))), its type $\{x : int, y : int, f : bool, \ldots\}$ becomes $(int \times_x (int \times_y (bool \times_f \alpha)))$. Constructing records this way is like building inhomogeneous lists, with a separate *cons* operator for each field. The system implementation can easily provide concrete syntax for our records and do the conversion to the representation. In fact, extensible records have been implemented in Isabelle/HOL as a result of the just presented work.

There are several distinguishing features of our encoding of extensible records in HOL, as compared to the set theoretic one presented earlier.

Most prominently, labels are not first class, but part of constant and type names (*x-field* and $\times_x$). Thus we can no longer refer directly to fields within the logic, "record r has field l" is not a HOL relation in our setting. Yet this does not prevent us to write generic functions $f\ \{x = a, y = b, \ldots\}$ that expect certain fields. This is actually the way we get record subtyping for free, in the guise of ordinary polymorphism. So we gain a lot by directly employing the HOL type system for record types.

Also, our records are not commutative: records $\{x = 3, y = 5\}$ and $\{y = 5, x = 3\}$ are different, even of incompatible types. So one has to ensure that records obey a canonical order of fields, which is not considered an actual limitation.

Furthermore, we do not provide a record merge operation. This would be basically concatenation of record types, requiring an associative operator. HOL with its free first-order type system cannot express this. We merely loose multiple inheritance because of this.

2.2.2 HOOL

We now introduce a logical environment HOOL that supports object-oriented concepts like classes, instantiation and inheritance. Our theory syntax will be similar to conventional object-oriented languages, like the one proposed in [23]. In this section we will only give some hints on how all of this can be implemented in terms of ordinary HOL declarations and definitions, see 2.2.3 for more details.

We use points, coloured points and rectangles as a running example. The root class *point* has x- and y-coordinates as fields, method *move* for moving points by a given offset and methods *reflect-X*, *reflect-Y*, *reflect-O* for reflecting them along the abscissa, ordinate, origin, respectively. Class *cpoint* adds a colour component to points. Class *rectangle* is a subclass of *cpoint* and specifies rectangles, which are determined by a reference point (bottom-left) together with the width and height. Rectangles are always in parallel to the x/y-axes. We also introduce a class *rectangle-hilite* of rectangles that set the colour to red when being moved.

Classes

To begin with the example, we define a root class *point*.

class $point =$

fields $x, y : int$

final methods

$reflect\text{-}O : \{x : int, y : int, \ldots\} \to \{x : int, y : int, \ldots\}$
$reflect\text{-}O = this.reflect\text{-}Y \circ this.reflect\text{-}X$

methods

$move : \{x : int, y : int, \ldots\} \to int \to int \to \{x : int, y : int, \ldots\}$
$move\ \{x, y, \ldots\}\ dx\ dy = \{x + dx, y + dy, \ldots\}$
$reflect\text{-}X : \{x : int, y : int, \ldots\} \to \{x : int, y : int, \ldots\}$
$reflect\text{-}X\ \{x, y, \ldots\} = this.move\ \{x, y, \ldots\}\ 0\ (-2 \cdot y)$
$reflect\text{-}Y : \{x : int, y : int, \ldots\} \to \{x : int, y : int, \ldots\}$
$reflect\text{-}Y\ \{x, y, \ldots\} = this.move\ \{x, y, \ldots\}\ (-2 \cdot x)\ 0$

specification

$move\ p\ 0\ 0 =^{x,y} p$ (1)
$(move\ p\ dx\ dy).x = p.x + dx$ (2)
$(move\ p\ dx\ dy).y = p.y + dy$ (3)
$this.move\ (reflect\text{-}X\ p)\ dx\ dy =^{x,y}$ (4)
$\quad reflect\text{-}X\ (this.move\ p\ dx\ (-dy))$
$reflect\text{-}Y\ (reflect\text{-}X\ p) =^{x,y} reflect\text{-}X\ (reflect\text{-}Y\ p)$
$reflect\text{-}X\ (reflect\text{-}X\ p) =^{x,y} p$
$reflect\text{-}Y\ (reflect\text{-}Y\ p) =^{x,y} p$
$reflect\text{-}O\ (reflect\text{-}O\ p) =^{x,y} p$

We refer to methods in two ways, written with or without a prefix *this*. This distinction plays a vital rôle for inheritance, but can be ignored at the moment. Note that we use a particular equality $=^{x,y}$ which expresses that two points coincide on the coordinates, but not necessarily on the remaining fields. To improve readability, correctness proofs are not shown here. Verification issues are discussed in Section 2.2.4.

Since we are within a functional setting, state-modifying methods are modeled as functions mapping states to states. To be more precise, methods do not operate on particular states but on arbitrary instances of a given state scheme.

Mutual dependencies of methods are acceptable as long as they are non-circular. Recursive definition of methods is not supported as a primitive. The user has to express this using appropriate operators from the underlying logic (e. g. well-founded recursion).

Objects, instantiation, and method invocation

Objects are *instantiated* from classes by specialisation. Instantiating some concrete object *MyPoint* from class *point* is achieved by specialising the state-space from $\{x : int, y : int, \ldots\}$ to $\{x : int, y : int\}$ and determining

the initial values for the coordinates. For instantiation we write $MyPoint = new\ point\ \{x = 3, y = 5\}$.

Method invocation is simply achieved by function application. For example, we can reset the object $MyPoint$ by $move\ MyPoint\ (-MyPoint.x)\ (-MyPoint.y)$.

Inheritance

Inheritance means being able to reuse code of superclasses in subclasses without explicit alteration. At first sight, this problem seems to be trivial just by duplicating code, but the problem is slightly more complicated. Consider, for example, the point methods, which operate on an x- and y-coordinate whereas the same methods (seen as methods of coloured points) have to operate on an extended state-space, which contains a colour field, too. Using extensible records we are able to write code for point methods generically such that the methods can operate on any state-space which contains at least x- and y-coordinates. Hence our implementation of the *point* methods can be used in a class of coloured points *cpoint* without alteration. This is what we achieve by the following definition:

$$\begin{array}{l}\textbf{datatype}\ colour = Red \mid Green \mid Blue\\ \textbf{class}\ cpoint = point + \textbf{fields}\ col : colour\end{array}$$

As suggested by above "+" notation, class *cpoint* includes all fields and methods from *point*.

Overriding

To continue with the example, we define a new class *rectangle*, adding fields w, h and method *area*. Reflecting a rectangle cannot be achieved by simply reflecting the reference point. When reflecting the bottom-left point along the x-axis it becomes the top-left point, so we have to subtract the height of the rectangle from its y-value to fix this. An analogous correction has to be performed for the reflection along the x-axis.

$$\begin{array}{l}\textbf{class}\ rectangle = cpoint\ +\\ \quad \textbf{fields}\ w, h : nat\\ \quad \textbf{methods}\\ \quad\quad area : \{x : int, y : int, col : colour, w : nat, h : nat, \ldots\} \to nat\\ \quad\quad area\ \{x, y, col, w, h, \ldots\} = w \cdot h\\ \quad \textbf{override methods}\\ \quad\quad reflect\text{-}X\ \{x, y, col, w, h, \ldots\} =\\ \quad\quad\quad this.move\ (point.reflect\text{-}X\ \{x, y, col, w, h, \ldots\})\ 0\ (-h)\\ \quad\quad reflect\text{-}Y\ \{x, y, col, w, h, \ldots\} =\\ \quad\quad\quad this.move\ (point.reflect\text{-}Y\ \{x, y, col, w, h, \ldots\})\ (-w)\ 0\end{array}$$

specification
$area\ (move\ \{x, y, col, w, h, \ldots\}\ dx\ dy) = area\ \{x, y, col, w, h, \ldots\}$

Apart from *reflect-X* and *reflect-Y* all methods and all lemmas of *cpoint* are inherited. At first sight it appears evident what we mean by saying "all other methods are inherited". But life is not as easy as it seems. Recall the definition of *reflect-O* in *point*: $reflect\text{-}O = this.reflect\text{-}Y \circ this.reflect\text{-}X$. On the one hand we have inherited this method, on the other hand we have overridden the methods *reflect-X* and *reflect-Y* in *rectangle*. If *this.reflect-X* and *this.reflect-Y* referred statically to the methods defined in *point* the method *reflect-O* would not behave as expected for rectangles. Instead, the references to *reflect-X* and *reflect-Y* in the inherited method *reflect-O* must refer dynamically to the redefined methods. In the following section we will have a closer look at this dynamic binding of methods which sometimes is also called *late-binding*.

Late-binding

Late-binding of methods is a powerful mechanism, making reuse of code very flexible. To back up this claim we extend rectangles by a class *rectangle-hilite*. The idea is that relocated rectangles are highlighted in red colour. Without late-binding of methods we would have to redefine *all* methods (except for *area*). The impact of these modifications on the correctness proofs would be disastrous: almost all proofs about rectangles would have to be repeated, quite redundantly though. Using late-binding of methods, the definition of *rectangle-hilite* is very simple because all methods relocating rectangles are defined directly or indirectly in terms of the generic *move*.

class *rectangle-hilite* = *rectangle* +

override methods
$move = (update\text{-}colour\ Red) \circ rectangle.move$

specification
$col\ (reflect\text{-}X\ \{x, y, col, w, h, \ldots\}) = Red$
$col\ (reflect\text{-}Y\ \{x, y, col, w, h, \ldots\}) = Red$
$col\ (reflect\text{-}O\ \{x, y, col, w, h, \ldots\}) = Red$

The fact that we can prove these properties of *reflect-X*, *reflect-Y* and *reflect-O* is remarkable. Without having redefined any of these methods, the change of the *move* method has been propagated automatically. This demonstrates that object-oriented verification really does work in our environment.

Now we have arrived at a point where we can clarify the distinction between those methods prefixed by *this* and those which are not. Methods prefixed

by *this* are late-bound and may change in subclasses whereas the others are fixed. For a better understanding of the distinction recall equation (4) from *point*:

$$this.move\ (reflect\text{-}X\ p)\ dx\ dy =^{x,y} reflect\text{-}X\ (this.move\ p\ dx\ (-dy))$$

This equation expresses that all implementations of the late-bound method *this.move* in subclasses are well behaved together with the particular implementation *reflect-X* of *point*. Expanding the definition of *reflect-X* — we cannot expand any definition of *this.move* since it is late-bound — in *point* yields:

$$\begin{array}{l} this.move\ (this.move\ p\ 0\ (-2 \cdot (y\ p)))\ dx\ dy =^{x,y} \\ \quad this.move\ (this.move\ p\ dx\ (-dy))\ 0\ (-2 \cdot (y\ (this.move\ p\ dx\ (-dy)))) \end{array}$$

Of course, there are implementations of *this.move* invalidating this equation. However, it is true for all implementations satisfying the equations for *move* given in *point*. Assuming that equations (1)–(3) hold for all implementations of *this.move* in subclasses, we can always show equation (4). This implies that (4) can be inherited in *rectangle-hilite* although method *move* has been overridden. Of course, we do not get all proofs for free. Since we have overridden *move*, we have to redo the proofs for all equations containing a particular implementation of *move* (without prefix *this*, that is).

2.2.3 Encoding

We now show that the object-oriented concepts presented in Section 2.2.2 are only a stone's throw away from a rigorous encoding in HOL.

States are represented as extensible records and methods as state transforming functions. As we have already seen, we can achieve inheritance simply by record subtyping. Things are getting much more complicated when taking late-binding into account. What makes it hard to model is that the semantics of late-bound methods changes relatively to the position in the inheritance hierarchy. Assuming different field types for different levels of the hierarchy, we can use *overloading* to achieve different meaning of methods in different contexts. Assuming different field types for different levels is no real restriction, since we can always enforce them by adding dummy fields.

Classes

First of all, the fields of any class definition become a record type definition:

$$\begin{array}{c} \textbf{record}\ point = \\ x, y : int \end{array}$$

Methods are more involved. The simplest method of *point* is *move*, because it is not late-bound.

First attempt One might try to realize method *move* in HOL directly as suggested in the *point* class definition:

$$\begin{array}{l} move : \{x : int, y : int, \ldots\} \to int \to int \to \{x : int, y : int, \ldots\} \\ move\ \{x, y, \ldots\}\ dx\ dy = \{x + dx, y + dy, \ldots\} \end{array}$$

The problem with this definition is that it is too generic. Since *move* is defined for *all* records containing x- and y-coordinates, we cannot override this definition in subclasses any more.

Second attempt To remedy this problem one might *declare* the method generically, but *define* it on concrete records only:

$$\begin{array}{l} move : \{x : int, y : int, \ldots\} \to int \to int \to \{x : int, y : int, \ldots\} \\ move\ \{x, y\}\ dx\ dy = \{x + dx, y + dy\} \end{array}$$

With this definition we are able to express overriding and late-binding. Overriding is achieved simply by defining *move* on a different concrete instance of the scheme $\{x : int, y : int, \ldots\}$, say on $\{x : int, y : int, col : colour\}$. To see, how we can achieve late-binding consider the definition of a method *reset* which sets points to the origin:

$$\begin{array}{l} reset : \{x : int, y : int, \ldots\} \to \{x : int, y : int, \ldots\} \\ reset\,\{x, y, \ldots\} = move\,\{x, y, \ldots\}\,(-x)\,(-y) \end{array}$$

Since we have given no definition of *move* on the extensible record type, its semantics and hence the semantics of *reset* is unspecified. Restricting the extensible record type to the concrete one $\{x : int, y : int\}$, determines a meaning as given by definition of *move*. Restricting it to a different concrete record may result in a different meaning, depending on the definition of *move* given there.

There is still a snag: we have ruled out inheritance. By defining *move* on a concrete record type we lose the ability to reuse code in subclasses.

The solution To achieve all (overriding, late-binding and inheritance), we define two constants *point.move* and *this.move* rather than a single constant *move*, allowing the character "." to be part of identifiers. The actual implementation of the *move* method in HOL is as follows:

$$\begin{array}{l} point.move, this.move : \\ \quad \{x : int, y : int, \ldots\} \to int \to int \to \{x : int, y : int, \ldots\} \\ point.move\,\{x, y, \ldots\}\,dx\,dy = \{x + dx, y + dy, \ldots\} \\ this.move\,\{x, y\} = point.move\,\{x, y\} \end{array}$$

Apart from *reflect-O*, the remaining methods are defined analogously. Since *reflect-O* is a *final* method, we have to guarantee that it cannot be overridden. Defining it on an extensible record type achieves this:

$$
\begin{array}{l}
point.reflect\text{-}O, this.reflect\text{-}O : \\
\quad \{x : int,\ y : int,\ \ldots\} \to \{x : int,\ y : int,\ \ldots\} \\
point.reflect\text{-}O = this.reflect\text{-}Y \circ this.reflect\text{-}X \\
this.reflect\text{-}O = point.reflect\text{-}O
\end{array}
$$

The need of two definitions for one method is no real problem for the user. These definitions can be generated automatically by some extra-logical system support.

Objects and instantiation

Instantiation is trivial in our framework. Just let $MyPoint = \{x = 3,\ y = 5\}$. The simplicity of instantiation stems from the fact that we generate both generic *class methods* and concrete *object methods* in classes. In a sense, we have anticipated instantiation already by the way we define classes.

Inheritance

Inheritance is just as simple as instantiation. For inheritance, all we have to do is specialise the class methods of the superclass to concrete object methods of the subclass. Class *cpoint* leads to the following definitions in HOL:

$$
\begin{array}{l}
\textbf{record}\ cpoint = point\ + \\
\quad col : colour \\
\\
\quad this.move\ \{x,\ y,\ col\} = point.move\ \{x,\ y,\ col\} \\
\quad this.reflect\text{-}X\ \{x,\ y,\ col\} = point.reflect\text{-}X\ \{x,\ y,\ col\} \\
\quad this.reflect\text{-}Y\ \{x,\ y,\ col\} = point.reflect\text{-}Y\ \{x,\ y,\ col\}
\end{array}
$$

Interestingly, we do not have to give a definition for *reflect-O* once more. Since *reflect-O* was defined for the scheme $\{x : int,\ y : int,\ \ldots\}$, its definition works equally well on the concrete type $\{x : int,\ y : int,\ col : colour\}$.

Since the methods have not been altered in *cpoint*, lemmas proved for points also hold for coloured points. Sticking to object-oriented terminology, we might say that the proofs are inherited. In a type-theoretic framework with explicit proof-terms this terminology fits perfectly well (see also [12]).

Overriding

Class *cpoint* serves as an example for inheritance, but it does not demonstrate overriding. Overriding is achieved simply by defining new methods. In case of class *rectangle* we define methods *rectangle.reflect-X* and *rectangle.reflect-Y*:

$$\begin{array}{l}
\textbf{record}\ \mathit{rectangle} = \mathit{cpoint} + \\
\quad w, h : \mathit{nat} \\
\quad \mathit{rectangle.reflect\text{-}X}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \ldots\} = \\
\qquad \mathit{this.move}\ (\mathit{point.reflect\text{-}X}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \ldots\})\ 0\ (-h) \qquad (5) \\
\quad \mathit{this.reflect\text{-}X}\ \{x,\ y,\ \mathit{col},\ w,\ h\} = \mathit{rectangle.reflect\text{-}X}\ \{x,\ y,\ \mathit{col},\ w,\ h\} \\
\quad \mathit{rectangle.reflect\text{-}Y}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \ldots\} = \\
\qquad \mathit{this.move}\ (\mathit{point.reflect\text{-}Y}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \ldots\})\ (-w)\ 0 \\
\quad \mathit{this.reflect\text{-}Y}\ \{x,\ y,\ \mathit{col},\ w,\ h\} = \mathit{rectangle.reflect\text{-}Y}\ \{x,\ y,\ \mathit{col},\ w,\ h\}
\end{array}$$

Late-binding

Class *rectangle-hilite* is a good example for late-binding of methods. Apart from late-binding, class *rectangle-hilite* is interesting because it introduces no new fields. Since we have identified class membership with field types, we have to tell the field types of *rectangle-hilite* and *rectangle* apart by adding an artificial field *dummy* of type *unit*. For simplicity we omit some obvious method definitions.

$$\begin{array}{l}
\textbf{record}\ \mathit{rectangle\text{-}hilite} = \mathit{rectangle} + \\
\quad \mathit{dummy} : \mathit{unit} \\
\quad \mathit{rectangle\text{-}hilite.move} : \\
\qquad \{x : \mathit{int},\ y : \mathit{int},\ \mathit{col} : \mathit{colour},\ w : \mathit{nat},\ h : \mathit{nat},\ \mathit{dummy} : \mathit{unit},\ \ldots\} \rightarrow \\
\qquad\quad \{x : \mathit{int},\ y : \mathit{int},\ \mathit{col} : \mathit{colour},\ w : \mathit{nat},\ h : \mathit{nat},\ \mathit{dummy} : \mathit{unit},\ \ldots\} \\
\quad \mathit{rectangle\text{-}hilite.move} = (\mathit{update\text{-}col}\ \mathit{Red}) \circ \mathit{rectangle.move} \qquad (6) \\
\quad \mathit{this.move}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \mathit{dummy}\} = \\
\qquad \mathit{rectangle\text{-}hilite.move}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \mathit{dummy}\} \qquad (7) \\
\quad \mathit{this.reflect\text{-}X}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \mathit{dummy}\} = \\
\qquad \mathit{rectangle.reflect\text{-}X}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \mathit{dummy}\} \qquad (8)
\end{array}$$

In this class, method *move* additionally sets the colour to *Red*. All methods defined in terms of *move* show the same effect, as can be seen by expansion of their definitions. Take for example method *this.reflect-X* (we abbreviate the term $\mathit{point.reflect\text{-}X}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \mathit{dummy}\}$ by Δ):

$$\begin{array}{ll}
\mathit{this.reflect\text{-}X}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \mathit{dummy}\} = & \text{by (8)} \\
\mathit{rectangle.reflect\text{-}X}\ \{x,\ y,\ \mathit{col},\ w,\ h,\ \mathit{dummy}\} = & \text{by (5)} \\
\mathit{this.move}\ \Delta\ 0\ (-h) = & \text{by (7)} \\
\mathit{rectangle\text{-}hilite.move}\ \Delta\ 0\ (-h) = & \text{by (6)} \\
((\mathit{update\text{-}col}\ \mathit{Red}) \circ \mathit{rectangle.move})\ \Delta\ 0\ (-h) &
\end{array}$$

Be aware, that *this.reflect-X* may have a completely different meaning on different state spaces.

2.2.4 Verification

Subsequently we investigate up to what extent object-oriented concepts, developed to structure programs, provide means to structure verification,

too. Since we have introduced two kinds of methods, class methods and object methods, we naturally expect two kinds of lemmas. In the end, though, we are only interested in those lemmas about object methods.

Object methods What distinguishes object methods and class methods, anyhow? There are two main characteristics for object methods: they are prefixed by *this* (which is merely a syntactic convention) and they are only defined on concrete records. Proving lemmas about object methods does not require any particular methodology. Take for example the following equation

$$this.move\,\{x, y\}\,0\,0 =^{x,y} \{x, y\}$$

which is immediately proven by rewriting. Proving lemmas on object methods directly, though possible in principle, is not very clever: we do not exploit object-oriented structuring principles for verification. We argue now that verification of class methods entails abstract and thus structured verification.

Class methods There are both late-bound and fixed class methods. Late-bound methods are prefixed by *this* (again, this is only a syntactic convention). More importantly, they are *only pre-declared*, without fixing a concrete definition yet, e.g $this.move : \{x : int, y : int, \ldots\} \to int \to int \to \{x : int, y : int, \ldots\}$. Fixed class methods are prefixed by class names, *declared and defined* on extensible record types and may use late-bound methods for definition. As an example consider:

$$\begin{array}{l} point.reflect\text{-}X : \{x : int, y : int, \ldots\} \to \{x : int, y : int, \ldots\} \\ point.reflect\text{-}X\,\{x, y, \ldots\} = this.move\,\{x, y, \ldots\}\,0\,(-2 \cdot y) \end{array}$$

For fixed methods not referring to late-bound methods we can prove lemmas directly. Take for example equation (1) with every occurrence of *move* replaced by *point.move* — we write (1)[*point.move*/*move*]. This lemma is immediately proven by rewriting. Since the lemma expresses a property for all state-spaces which contain at least x- and y-coordinates, this lemma holds in all subclasses as long as method *point.move* is inherited. The same holds for the next three equations. By restricting the state-space to concrete records, we get the corresponding lemmas for object methods for free (by HOL type instantiation).

What happens if fixed methods refer to late-bound methods? Since late-bound methods are only declared, we canot expect non-trivial lemmas to hold for such methods. To prove interesting lemmas we have to assume properties of the late-bound methods. For class definitions we apply the convention that the lemmas are ordered by position and (implicitly) have the preceding lemmas as assumptions. To be precise, the n-th lemma of

point is translated to the following formula in HOL (where (i) stands for the i-th lemma of *point*, and $[this.m/m]$ or $[point.m/m]$ for prefixing all non-prefixed methods by *this* or *point*, respectively):

$$\bigwedge_{i=1}^{n-1}(i)[this.m/m] \Rightarrow (n)[point.m/m]$$

The question arising immediately is how to get rid of assumptions. On the one hand, we cannot discharge them in classes (which are basically abstract theories). On the other hand, assumptions can be eliminated in any concrete instance, where class methods are specialised to object methods.

Finally, we explain observational equality $=^{x,y}$ which is defined as follows:

$$\begin{array}{l} =^{x,y} : \{x : int,\, y : int,\, \ldots\} \to \{x : int,\, y : int,\, \ldots\} \to bool \\ p =^{x,y} q = (x\,p) = (x\,q) \wedge (y\,p) = (y\,q) \end{array}$$

We use observational equality $=^{x,y}$ rather than ordinary equality for specification in cases when we only want to fix the meaning of methods on the coordinates. To see why this is more appropriate than actual equality, recall the definition of *move* in *rectangle-hilite*: for full equality, equation (1) would no longer hold in *rectangle-hilite* because the *col* field is manipulated there.

Let us leave our running example and see what we have achieved. We can specify non-late-bound class methods generically and thus inherit the proofs in subclasses immediately. To deal with late-bound methods we have to add assumptions to the equations to be proven. But we know that in all subclasses we can discharge the assumptions.

What happens if we override methods? Depending on which kind of method is used we get different consequences. If a method is late-bound, we cannot use any information about its implementation for the correctness proof and hence we can inherit the proof even if the method is overridden. If a method occurs non-late-bound at least once, we have to perform a new proof.

So to cut a long story short, appropriate use of late-bound methods does not only cater for flexible reuse of code, it also provides a mechanism for generic and thus reusable correctness proofs.

2.3 Memories and References

The quotation at the beginning of the introduction, which goes back to Wilhelm von Ockham (1285 - 1349), was already a sign that our framework Logos can not do without an explicit model of memories and references. Indeed, we will see later that memories and references are fundamental for

Logos. Since these constructs are not provided by HOL itself, we have to define a theory of memories and references ourselves.

We strictly distinguish between objects and references. Formalising these notions, it is intuitive to define objects and references side by side in memories and to define standard operations on these memories as allocating new objects and dereferencing references. Since we are working in a typed environment, we define memories for typed objects, entailing polymorphic definitions for the theory of references.

As we intend to use references in different contexts we have to take care of great flexibility. One way to achieve flexibility is by polymorphic definitions, which you may call the *syntactic way*. Another way is to ensure well-behaviour by axiomatic type classes, that is logical formulas, which you may call the *semantic way*. These logical formulas – if applied properly – gives rise to abstract theories which allow for implementation-independent reasoning. Semantic restrictions may be expressed by predicates which have to be dealt with explicitly in formulas.

An axiomatic type class simply is a class of types that all meet certain axioms. Thus, type classes may also be understood as type predicates i.e. abstractions over a single type argument α. Class axioms typically contain polymorphic constants that depend on this typ α. The characteristic constants behave like operations assocciated with the carrier α. Axiomatic type classes are incorporated in Isabelle/HOL (see [46]) with two restrictions: axiomatic type constructor classes as well as multiple type arguments are not supported.

We will use axiomatic type classes to define an abstract theory of references. As our theory of references ought to be able to store objects of any type τ, memories are polymorphic in this type τ. The axiomatic type class we define for references expresses certain important properties that have to be shared by all implementations of memories and references.

To be precise, though, axiomatic type classes are not sufficient to express our theory. Since memories and references are meant to be polymorphic we have to apply *axiomatic type constructor classes*. Let's see why. As we have said, our memories are to be polymorphic and so are the references. Formally, this means, that the types of memories are formed by type constructors with arity one (i.e. one type parameter: the type of the objects). The same holds for references, too. Focusing on one of the most important operations *dref* for dereferencing references, we see why constructor classes are necessary. The type of $dref : \tau\ Ref \to \tau\ memory \to \tau\ option$ mentions memories, references and also objects (please ignore type constructor *option* for the moment). Modelling this operation with type classes would require the type of the memory $\tau\ memory$ and the type of the references $\tau\ Ref$ to be replaced with type variables α and β. The resulting type $\beta \to \alpha \to \tau\ option$ breaks two important rules, though. First, we have lost

the information that the memories and references are polymorphic in τ and second we have type variables (namely τ) in addition to the type variables α and β of the type class. The solution to this problem is, as already mentioned, to use constructor classes [15, 29] rather than type classes. Constructor classes are, as the name already suggests, polymorphic in type constructors rather than types themselves. For our references this means that we have to define a constructor class polymorphic in *memory* and *Ref*. Any instance of this constructor class has to provide two concrete type constructors with respective operations and proofs that these operations meet the specification given in the axiomatic constructor class.

In order to express certain properties of references we have to define the signatures of the operations involved. Signatures are defined in our setting by declaring constants with the signatures as types. We call the declared constants *abstract operations* and postulate important properties by means of an axiomatic type constructor class $Ref_{axclass}$. The intended behaviour of these abstract operations is described in the following. Please note that objects are never addressed directly; they are only accessed by means of references.

- **newmemory**: Creates a new, empty memory – to account for easy composition it is modeled as a function with input of type unit
- **allocate**: Allocates a new object in the memory – the new object is added to the objects and a new reference is created and added to the references
- **freshref**: Returns the first fresh i.e. yet unused reference – as a special case it may be used to compute the reference to a newly allocated object
- **dref**: Returns the object denoted by the reference if exists; raises error otherwise
- **lift**: Overwrites the value of an object (denoted by a reference) by a new value (depending on the old value of the object)
- **store**: Overwrites the value of an object (denoted by a reference) by a new value (independently of the old value of the object)
- **objects**: Computes the set of all active objects – only those objects are included that are denoted by references; all other objects are of no relevance and are expected to be wiped off by garbage collection
- **objects_all**: Computes the set of all objects in a memory – including objects that are no longer denoted by references

- **references**: Computes the set of all references
- **references_obj**: Computes the set of all references that are part of some object

Operation *dref* deals with errors explicitly. To do so a standard datatype called *option* is used.

$$\textbf{datatype}\ \alpha\ option = None \mid Some\ \alpha$$

Any element of type $\alpha\ option$ is either an explicit error (represented by constructor *None*) or some valid value of type α (represented by constructor *Some*). Datatype *option* is equipped with a set of standard operations:

$$\begin{array}{l} bind : \alpha\ option \to (\alpha \to \beta\ option) \to \beta\ option \\ bind\ None\ f = None \\ \quad \mid \quad (Some\ x)\ f = f\ x \\ map : (\alpha \to \alpha) \to \alpha\ option \to \alpha\ option \\ map\ f\ None = None \\ \quad \mid \quad f\ (Some\ x) = Some\ (f\ x) \\ peel_{option} : \alpha\ option \to \alpha \\ peel_{option}\ None = arbitrary \\ \quad \mid \quad (Some\ x) = x \end{array}$$

For convenience we abbreviate $bind\ o\ (\lambda\, x.\ f\ x)$ as $x := o; f\ x$. Please note that we will also use operator ";" for ordinary functional composition (see footnote in Example 3). This coincidence is intended since in both cases operator ";" composes elements. In the just defined case, though, operator ";" also takes care of error handling.

In the following we will ignore error handling quite often. Explicit error handling by type *option* inflates the definitions (even if monads were used – see [45, 44]) and thus is opposed to a comprehensive presentation. Nevertheless, all definitions could be corrected easily.

Above informal specification of the operations translates directly to signatures (i.e. constant declarations).

Definition 1 (Signature of references) *The general theory of references contains operations with following types:*

$$\begin{array}{ll} new_{memory} & : unit \to \tau\ memory_{var} \\ alloc & : \tau \to \tau\ memory_{var} \to \tau\ memory_{var} \\ freshref & : \tau\ memory_{var} \to \tau\ Ref_{var} \\ dref & : \tau\ Ref_{var} \to \tau\ memory_{var} \to \tau\ option \\ lift & : (\tau \to \tau) \to \tau\ Ref_{var} \to \tau\ memory_{var} \to \tau\ memory_{var} \\ store & : \tau \to \tau\ Ref_{var} \to \tau\ memory_{var} \to \tau\ memory_{var} \\ objects & : \tau\ memory_{var} \to \tau\ list \\ objects_{all} & : \tau\ memory_{var} \to \tau\ list \\ references & : \tau\ memory_{var} \to (\tau\ Ref_{var})\ list \\ references_{obj} & : \tau\ Ref_{var} \to (\tau : myreferences_{axclass})\ memory_{var} \to (\tau\ Ref_{var})\ set \end{array}$$

Please note that $memory_{var}$ and Ref_{var} are variables standing for type constructors. The signatures are thus polymorphic in these type constructor variables.

In the just listed constant declarations we have used an axiomatic type class $myreferences_{axclass}$ that we have not introduced so far. This axiomatic type class is required since operation $references_{obj}$ does not make sense for all types τ. To be implementable, we have to know how to extract the references from objects of type τ. Of course, we also have to know how to add, remove and modify these references. Axiomatic type class $myreferences_{axclass}$ provides operations to deal with references contained in objects of type τ.

Definition 2 (Signature of types with references)

$$\begin{array}{ll}
myreferences & : \tau\ memory_{var} \to \tau \to (\tau\ Ref_{var})\ list \\
myreferences_{add} & : \tau\ memory_{var} \to \tau\ Ref_{var} \to \tau \to \tau \\
myreferences_{remove} & : \tau\ memory_{var} \to \tau\ Ref_{var} \to \tau \to \tau \\
myreferences_{map} & : \tau\ memory_{var} \to (\tau\ Ref_{var} \to \tau\ Ref_{var}) \to \tau \to \tau
\end{array}$$

In order to define the type class class, we have to state the properties to be satisfied. These properties are expressed as predicates over the just listed functions.

Definition 3 (Axiomatic type class of types with references) *For propositions (1) and (2) assume $r \in references\ m$. For propositions (4) and (5) assume $r \notin references\ m$.*

axclass $myreferences_{axclass} =$

$$\begin{array}{ll}
myreferences\ (myreferences_{add}\ m\ r\ o) = (myreferences\ m\ o) \cup \{r\} & (1) \\
myreferences\ (myreferences_{remove}\ m\ r\ o) = (myreferences\ m\ o) - \{r\} & (2) \\
myreferences\ (myreferences_{map}\ m\ f\ o) = \{f\ x \mid x \in myreferences\ m\ o\} & (3) \\
myreferences\ (myreferences_{add}\ m\ r\ o) = myreferences\ m\ o & (4) \\
myreferences\ (myreferences_{remove}\ m\ r\ o) = myreferences\ m\ o & (5)
\end{array}$$

We are now ready to define an axiomatic type constructor class for our theory of references.

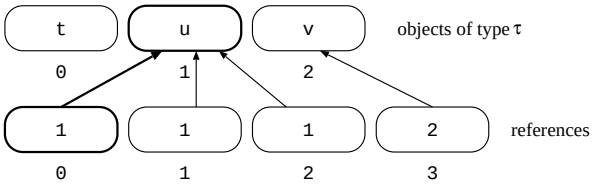
Definition 4 (Axiomatic type constructor class of references) *The axiomatic type constructor class $references_{axclass}$ is polymorphic in the type constructors $memory_{var}$ and Ref_{var}. It provides a list of properties shared by all theories of references.*

axclass $references_{axclass} =$

$$\begin{array}{ll}
(new_{memory}\ ()).references = \{\} \wedge (new_{memory}\ ()).objects_{all} = \{\} & (1) \\
m.objects \subseteq m.objects_{all} & (2) \\
m.objects = \{o \mid \exists r \in m.references.\ dref\ r\ m = Some\ o\} & (3) \\
dref\ m.freshref\ (alloc\ o\ m) = Some\ o & (4) \\
r \in m.references \Rightarrow dref\ r\ m \in \{Some\ x \mid x \in m.objects\} & (5)
\end{array}$$

In order to account for a complete definition of references we have to define concrete type constructors *memory* and *Ref*, implement the operations (i.e define them for the concrete type constructors) and show that the operations meet the specification of $references_{axclass}$.

Memories are defined intuitively as pairs consisting of lists of objects and lists of references (represented as natural numbers). At each position of a reference-list a reference (a natural number) is stored which denotes the position of the referenced object in the objects-list. Assume e.g the reference (natural number) 1 being stored in the head of the reference list and object 1 holding value u. You may read this as "reference 0 points to object 1 which currently holds value u".



Taking this idea as basis, it is easy to define polymorphic memories consisting of object-lists and reference-lists.

Definition 5 (Memories) *Memories are polymorphic in the type of objects τ. Objects and references are stored in τ memory as lists.*

$$\begin{array}{l}\textbf{\textit{record}}\ \tau\ memory = \\ \quad objs : \tau\ list \\ \quad refs : nat\ list\end{array}$$

Lists over type τ can be seen as finite functions from *nat* to τ. This is stressed by function $nth : nat \to \tau\ list \to \tau$ which computes the nth element in a list (hence the name). Since any function in our setting is total, function *nth* is defined for any natural number even if the number exceeds the length of the list. If the argument exceeds the length of the list an arbitrary element of type τ, called *arbitrary*, is returned. You may wonder, why we have not chosen functions directly to represent objects and references. Indeed this alternative definition would have been possible leading to the following definition:

$$\begin{array}{l}\textbf{record}\ \tau\ memory = \\ \quad objs : nat \to \tau \\ \quad refs : \tau\ Ref \to nat\end{array}$$

The empty memory then would be defined as $\{objs = \lambda\, x.\ arbitrary,\ refs = \lambda\, x.\ arbitrary\}$ and function $dref\ r\ \{objs,\ refs\} = (refs;\ objs)\ r$ would only amount to functional composition of *refs* and *objs*. All other operations on references could be defined easily as well. Why haven't we chosen this representation then instead of the list representation? First of all, it would indeed

have been possible to use this representation instead. The major drawback of the functional representation is that it also admits infinite memories whereas memories formed by lists are finite by definition. We will see in Section 3.1.2 why finite memories are crucial for our approach.

Finiteness of lists is due to construction of the datatype – the finiteness is thus *syntactic*, if you like. Ensuring that functions are finite requires logical predicates, hence you may call this solution *semantic*. It is folklore that dealing with syntactic restrictions is much easier when working in a computer-related and hence rigorously formal environment whereas semantic restrictions are much easier to handle when dealing in a non-rigorous intuitive environment as in ordinary mathematics. As we have set our environment to be rigorously formal, the syntactic approach and hence the list representation is more convenient.

The central idea behind references is that they store the locations of the referenced objects. It is natural to represent the locations as natural numbers (just as the memory cells in a computer are numbered). We follow this standard approach with a subtle, yet striking difference. For each type τ we define a τ-indexed isomorphic copy of the natural numbers. Instead of representing references to objects of type τ by natural numbers, we represent these references by τ-indexed isomorphic copies of the natural numbers.

Definition 6 (References)

$$\textbf{\textit{datatype}}\ \tau\ Ref = Ref\ nat$$

Datatype *Ref* induces a simple function $peel_{Ref}$ to peel off constructor *Ref* (which we often skip to improve readability):

$$\begin{aligned} peel_{Ref} &: \tau\ Ref \to nat \\ peel_{Ref}\ (Ref\ n) &= n \end{aligned}$$

At first sight datatype *Ref* seems overkill, but if you look at it more closely you will find – besides the improved readability – a quite powerful and surprising effect. With this simple trick references are taking advantage of overloading (writing polymorphic functions that behave differently on references of different types). Take for example a function $add\text{-}ref : \tau\ Ref \to \tau\ Ref \to \tau\ Ref \to \tau\ memory \to \tau\ memory$ which "adds" the objects denoted by the first two references and stores the result in the third reference. For different types you may implement *add-ref* differently. Taking the natural numbers as type instances you might add the numbers whereas taking lists of natural numbers you might append the lists.

Since we have defined references as axiomatic type classes, the whole development (of course except for the definitions of the operations and

the proofs that these operations meet the specification of $references_{axclass}$) carries over to the functional representation – if required.

Once we have determined the representation of the memories it is up to us to define operations and to prove that they meet the specification of $references_{axclass}$. First the operations. Given the concrete representation of memories *memory*, the definition of the operations is straightforward and requires almost no explanation.

There is one issue, though, that requires attention: objects themselves may contain references and these references need not be well-formed (in the sense that they may refer to non-existing objects). There are two ways of dealing with this problem.

First is to ensure that only those objects are allocated that are well-formed. But even then you can not be sure if objects are stored by means other than allocation – hence you can not be completely sure that all objects are well-formed. The only way to render this approach feasible was to add a so called "closed world" assumption restricting storage of objects to a well defined set of operations and disallowing any other means to store objects. Again, this approach is *semantic* and thus inconvenient for a rigorously formal treatment. It will come as no surprise that we follow a more *syntactic* approach.

Second is to allow non-well-formed references in objects but to ignore those references when observing memories. The idea for this approach stems from process theory (see [21] for a standard text book). The state of the system (in our case the memory) is assumed being a black box – don't bother which rubbish is contained inside! All you can see is what predefined operations turn visible. In case of memories, ill-formed references may be contained within objects but operation $references_{obj}$ only enumerates those references that are well-formed. As long as you observe the black box only by means of operation $references_{obj}$ you will not notice if objects are ill-formed internally at all.

Taking this idea, we define the operations on references whose signatures we have listed above. Additionally to these operations we define an auxiliary function *cell* which returns the number of the cell that a reference is pointing to – if exists; otherwise it raises an error.

$$
\begin{array}{l}
cell : \tau\ Ref \to \tau\ memory \to nat\ option \\
cell\ (Ref\ x)\ m = \textbf{if}\ length\ m.refs \leq x\ \textbf{then}\ None \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ Some\ (nth\ x\ m.refs)
\end{array}
$$

The operations on references are defined for type constructors *memory* and *Ref* as follows:

Definition 7 (Operations on references)

$$
\begin{array}{lcl}
new_{memory}\,() & = & \{objs = [],\ refs = []\} \\
alloc\ o\ m & = & \{m.objs{+}{+}o,\ m.refs{+}{+}(length\ m.objs)\} \\
freshref\ m & = & Ref\ (length\ m.refs) \\
dref\ r\ m & = & x := cell\ r\ m; \\
 & & \textbf{\textit{if}}\ length\ m.objs \leq x\ \textbf{\textit{then}}\ None \\
 & & \qquad\qquad \textbf{\textit{else}}\ Some\ (nth\ x\ m.objs) \\
lift\ f\ r\ m & = & m\{objs := update_{nth}\ (cell\ r\ m)\ (f\ (dref\ r\ m))\ m.objs\} \\
store\ o & = & lift\ (\lambda x.\ o) \\
objects\ m & = & \{o \mid x < length\ m.refs\ \wedge\ dref\ (Ref\ x)\ m = Some\ o\} \\
objects_{all}\ m & = & \{o \mid o \in m.objs\} \\
references\ m & = & \{Ref\ x \mid x < length\ m.refs\ \wedge\ dref\ (Ref\ x)\ m = Some\ o\} \\
references_{obj}\ r\ m & = & myreferences\ m\ (dref\ r\ m)
\end{array}
$$

Function $update_{nth} : nat\ option \to \alpha\ option \to \alpha\ list \to \alpha\ list$ updates the nth position of a list with a new element – provided the position and the new element are error-free. Otherwise it leaves the list unchanged. The definition of $update_{nth}$ is a standard list definition and thus omitted.

As a last step we have to establish that the operations on references meet specification $references_{axclass}$.

Lemma 1 (Operations on references meet $references_{axclass}$) *The operations on references defined for type constructor memory meet specification $references_{axclass}$.*

This lemma was the last missing bit to establish that type constructor *memory* is an instance of axiomatic type constructor class $references_{axclass}$.

Definition 8 (Instance of $references_{axclass}$) *Type constructors memory and Ref are instances of axiomatic type constructor class $references_{axclass}$.*

$$\textbf{\textit{instance}}\ memory,\ Ref\ \textbf{\textit{of}}\ references_{axclass}$$

This definition entails that the signature of references with type $\tau\ memory$ replacing $\tau\ memory_{var}$ and $\tau\ Ref$ replacing $\tau\ Ref_{var}$ (see Definition 1) can be used freely in terms – as long as type τ is an instance of type class $myreferences_{axclass}$. The same holds for the proofs, too. Due to this instance declaration we can be sure that the operations meet the specification given in axiomatic type constructor class $references_{axclass}$ (see Definition 4) including all of its consequences.

Please note that axiomatic type constructor classes are (as already mentioned) not part of Isabelle/HOL. The definitions of this section have to be understood as meta-definitions that can be instantiated with true HOL definitions (defining memories and references directly as operations without the abstraction of axiomatic type constructor classes).

Chapter 3

Framework Logos

In order to give a formal underpinning of authorisation in groupware, we ought to have at least some understanding what groupware is. To this end, we first develop some thoughts on this issue and then show how they influence the design rationale of our authorisation model.

As a result of modern communication technologies, information can be spread freely over the internet, at the risk of far too many people getting access to highly sensitive information. Even if all recipients were affiliated with the same organisational entity, most information was not allowed to exceed a clearly defined set of recipients. In personal meetings only a few people are involved and each person decides individually whom to talk to and thus whom to share information with. Deciding in the internet whom to share information with and guaranteeing that people not concerned are excluded from information is a non-trivial task. The only way to restrict access is by specialised programs (so–called "groupware") which control the flow of information.

Although groupware systems differ in many details, there are two major questions to be answered by any groupware system:

- "Which person may perform which kind of action?" and
- "Which person may gain access to which kind of information?"

Obviously it would be most flexible to determine access rights on an individual basis as far as documents and persons are concerned. This solution, though, would by no means be practical. On the one hand, the set of recipients may be considerably large and may even change over time. On the other hand the author may not even know of many of the recipients. The decision to address a particular person is often based on the person's position in the organisational hierarchy, whose line–up is subject to permanent changes. A groupware system thus can only be supportive if it dependably

models the organisation's current structure and access rights are defined for sets of people[1] rather than individual persons.

Like any other computer program, groupware needs to model reality internally by data structures. In case of groupware, organisational structures are represented as so-called *groups*. Membership in a group expresses affiliation with a particular substructure. In the real world, the overall structure of an organisation is controlled by a small, highly trusted set of people and so is the graph of groups in a groupware system, which is often called the name and address book (NAB). The model of an organisational structure in a groupware system is correct and complete if each person (to be concise the internal representation of a person) is member of a group if and only if in the real world the person is affiliated with the organisational substructure represented by the group.

So far we have only been talking about organisational structures but we have not mentioned the tasks performed within such structures. Each task (which we will later call database) possesses a task manager (the manager of the database) which oversees the task. Of course there a number of responsibilities within a task. Although it would, at least in principle, be possible to represent these responsibilities as groups, such a choice would not be reasonable: the task manager may not have the right to change the group-structure relevant for the task. Letting each task manager change the overall group structure, though, would violate an important principle in system's design: modularity. Local changes may only affect adjacent areas but not the whole system. In order to deal with local responsibilities, these are represented as so-called *roles*. The task manager assigns subjects or groups to roles in a database. This assignment is often also called access control list (ACL). Roles then are local to a particular task and therefore do not affect the remaining system.

In the following section we will introduce a mathematical model which we call Logos and which serves to express and validate authorisation issues in groupware systems and applications. Groupware systems modelled by Logos consist of a number of databases and of a single name and address book (NAB). In the NAB all registered subjects are stored together with their assignments to groups. The scope of the NAB extends over all databases. Each database is maintained by a database manager who controls the access control list (ACL). The ACL is the place where access rights are determined locally for this particular database. Access rights are assigned to groups by detour of roles. Besides the ACL a database also comprises a data space, which holds a collection of (possibly heterogeneous) data.
The structure of the groupware systems under consideration (and thus of

[1]We will speak of "subjects" instead of "people" subsequently since rights might also be assigned to machines (servers)

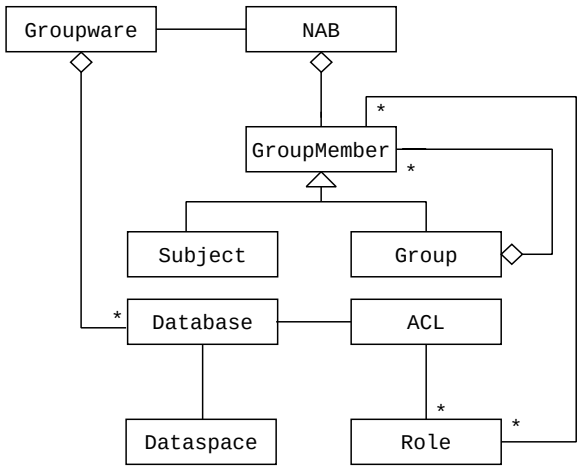


Figure 3.1: Logos framework (UML diagram)

the framework Logos) is depicted in Figure 3.1 as an UML diagram which coincidentally serves as a road map through this chapter. In the following we describe the components in more detail and show how to model them in Logos.

3.1 Groups

Traditional authorisation models control which *subjects* may perform which *operations* on which *objects*. If a subject may perform a particular operation the subject is said to have the *right* to execute the operation. In applications it would be too inflexible to define access rights for subjects individually. Therefore subjects are summarised to *groups*, i.e. sets of subjects that have equal rights. Rights are then assigned on the level of groups rather than the level of subjects. Groups may only consist of a single subject, hence – at least from a theoretical point of view – there is no need to define rights on the level of subjects, too. Subjects are assigned to groups in the name and address book (see Section 3.2.7). This assignment applies to all databases. In the following we traverse all of these issues starting with subjects.

3.1.1 Subjects

For Logos the names of the subjects are of no relevance (just as it does not make any difference if subjects are persons or objects as e.g. servers). One might simply use the natural numbers as an infinite set of names. We even go a step further and do not even assume that the set of subjects is infinite. We simply assume that subjects form a type.

Definition 9 (Subjects)

$$\textbf{\textit{type}}\ subject$$

For this section we use a simple example to illustrate the definitions. Though it is only a toy example, it is complex enough to demonstrate important features.

Example 1 (Subjects)

$$\textbf{\textit{datatype}}\ subject = Harry \mid Tom \mid Dick \mid Peter \mid Jenny$$

Groups are at the heart of Logos. In order to model hierarchic organisational structures, the obvious thing to do is to impose some structure on the groups in the sense that groups recursively may contain subgroups i.e. the groups themselves form a hierarchical structure. Of course, hierarchical group structures are not new in literature (e.g. see [41]). The proposed solutions differ less in their intentions than in their realizations and in the degree of their formalism.

One would expect the most natural definition of groups to be the following (the definition is taken from [41]):

- A (single) user is a user group.
- A set of user groups is a user group.

This definition can be translated directly into a datatype[2] definition (taking lists of user groups to form new user groups rather than sets of user groups, which does not make much of a difference since we are not interested in infinite sets of subgroups anyway).

Definition 10 (Groups - alternative I)

$$\textbf{\textit{datatype}}\ group = User\ subject \mid Subgroups\ (group\ list)$$

Indeed, this definition allows for hierarchical group structures as the following example demonstrates.

Example 2 (Groups – alternative I)

$$h = User\,Harry, \quad t = User\ Tom, \quad p = User\,Peter, \quad j = User\,Jenny$$
$$g_1 = Subgroups\,[h, t], \quad g_2 = Subgroups\,[p, g_1], \quad g_3 = Subgroups\,[j, g_1]$$

[2]Datatypes are a well-known definition principle in typed functional programming languages as ML, Haskell or Gofer (see [32, 13, 14]). Take for example the natural numbers which are defined as: **datatype** $nat = Zero \mid Succ\,nat$. This definition introduces a new type called "nat" whose elements are only formed by the following constant or function, respectively: $0 : nat$ and $Succ : nat \to nat$.

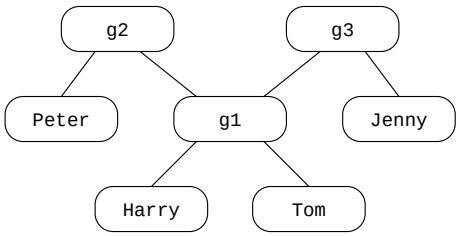


Figure 3.2: Group structure – alternative I

Problems arise when changing this group structure dynamically. See what happens if we try to add Dick to group g_1. First of all we would define a generic function addmember for adding new members to groups. Applying function addmember Dick to g_1 yields a new group say g_1', but this application leaves the definitions of g_2 and g_3 (and of course also g_1) unaffected.

$$addmember : subject \to group \to group$$
$$addmember\ p\ g = Subgroups\ [g,\ User\ p]$$

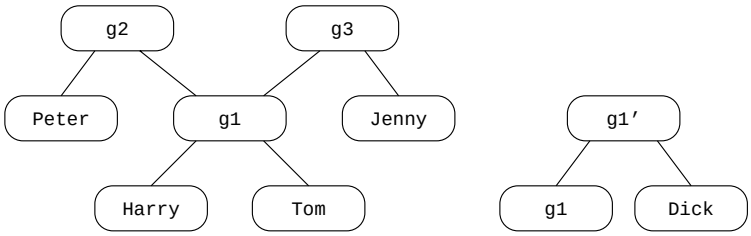


Figure 3.3: Group structure – alternative I (Dick added to g_1)

Even if we would update these definitions

$$g_2' = Subgroups\ [p,\ g_1'], \quad g_3' = Subgroups\ [j,\ g_1']$$

we might not be done. Assume any other group containing g_1 as subgroup. Its definition would have to be updated, too. □

Clearly, this is in conflict with modularity. The necessity to lift several definitions entails the danger of inconsistencies in case of any lifting being forgotten. The original structure, which establishes that g_1 is a subteam of both g_2 and g_3, may not be preserved.

As we have seen, the presented model of groups violates one fundamental paradigm in system design: **modularity**. In order to account for a reasonable model of dynamic group structures this deficiency has to be remedied. The solution to the problem suggests itself: references. To this end, the

definition of groups only requires a minor change. The list of subgroups has to be exchanged with a list of references to subgroups.

Definition 11 (Groups - alternative II)

$$\textbf{\textit{datatype}}\ group = User\ subject \mid Subgroups\ ((group\ Ref)\ list)$$

Let's recall above example and see how to model it with the renewed definition of groups. Because we have incorporated references to groups rather than groups themselves, we have to deal with memories[3] to store groups and references.

Example 3 (Groups – alternative II) *The group memory mem is constructed by the term*[4]

$$mem = new_{memory};\ alloc\ h;\ alloc\ t;\ alloc\ g_1;\ alloc\ p;\ alloc\ g_2;\ alloc\ j;\ alloc\ g_3$$

using the following abbreviations:

$$h = User\ Harry, \quad t = User\ Tom, \quad p = User\ Peter \quad j = User\ Jenny$$
$$g_1 = Subgroups\ [Ref\ 0,\ Ref\ 1], \quad g_2 = Subgroups\ [Ref\ 3,\ Ref\ 2],$$
$$g_3 = Subgroups\ [Ref\ 5,\ Ref\ 2]$$

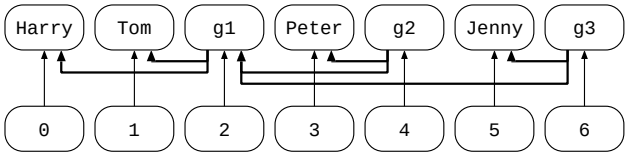


Figure 3.4: Group structure – alternative II

Let's see what happens now, if we try to add Dick to group g_1. Again we define a generic function

$$\begin{aligned} &addmember : subject \to group\ Ref \to group\ memory \to group\ memory \\ &addmember\ p\ r\ =\ alloc\ p; \\ &\qquad\qquad \lambda\ m.\ alloc\ (dref\ r\ m)\ m; \\ &\qquad\qquad \lambda\ m.\ store\ (Subgroups\ [(freshref\ m) - 2,\ (freshref\ m) - 1])\ r\ m \end{aligned}$$

to add new members to groups. Adding Dick to g_1 transforms the previous memory into a new memory $mem' = addmember\ Dick\ (Ref\ 2)\ mem$ as indicated in Figure 3.5.

[3] To form valid group memories we had – strictly speaking – to show that type *group* is an instance of axiomatic type class $myreferences_{axclass}$. We postpone this issue for a while (to page 45) where we discuss it in more detail.

[4] We sometimes write $f; g$ instead of $g \cdot f$ to ease readability. As for new_{memory}, we sometimes also write $x; f$ instead of $f(x)$.

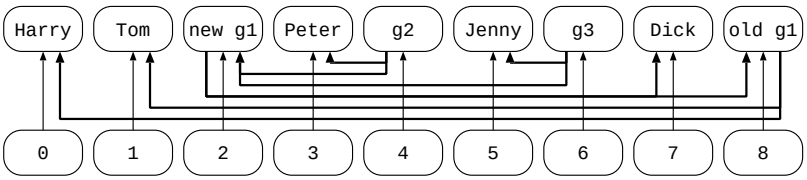


Figure 3.5: Group structure – alternative II (Dick added to g_1)

Although we are working in a functional environment, this model allows for invariable group names that remain constant even if the group structure was modified. Take for example group g_1 *and define*[5] $g_1 = dref\,(Ref\ 2)$. *We then can access this group in mem or mem′ uniformly:* $g_1\,mem$ *or* $g_1\,mem'$, *respectively.* □

From a theoretical point of view the new definition of groups with references is satisfactory. It remedies the deficiency inherent in the original definition: lack of modularity. Take function *addmember* as an example. Applying the function to some subject, say Dick, yields a function that is entitled to add Dick (if desired) to any group regardless its structure or occurrence in the group structure.

From a more practical point of view, though, *addmember* operates ponderously. It is not at all intuitive why this operation needs to allocate two new groups (one for the subject to be added and one for the former value of the group) just to add a subject to a group. In fact, the definition of groups can be adapted to account for a more intuitive behaviour.

Definition 12 (Groups)

$$\begin{aligned} \textbf{\textit{datatype}}\ group\ =\ & Empty\ \mid \\ & AddMember\ subject\ group\ \mid \\ & AddSubgroup\ (group\ Ref)\ group \end{aligned}$$

In some applications it is desirable to have a number of administrators affiliated with a group. The administrators' task is to administrate the group (as the name already suggests). Administrated groups can be captured quite easily by a minor generalisation of the just given definition. In the sequel we will always found on administrated groups rather than ordinary groups. All definitions can be adapted quite easily to account for non-administrated groups.

We use the same names for the type and constructors as for groups without administrators. In concrete applications it should be clear from the context which definition was meant.

[5]To be precise, we have used up name g_1 already. Consistently renaming the previous occurrences of g_1 would solve the naming conflict.

Definition 13 (Administrated Groups)

$$\begin{aligned} \textbf{\textit{datatype}}\ group\ =\ & Admins\ (group\ Ref)\ list\ \mid \\ & AddMember\ subject\ group\ \mid \\ & AddSubgroup\ (group\ Ref)\ group \end{aligned}$$

You may wonder why administrators need to form groups rather than just lists of subjects. Imagine a group of highly trusting people (as the project-team in project "Daidalos" – see Section 5.2.1) who may want to achieve highest flexibility, assigning each member administrator rights. This can be realized as a group with self-referring administrator group (i.e. the reference to the administrator group points to the group itself). Once a new member is added to the group, it is assigned administrator rights automatically – avoiding painful inconsistencies.

Taking lists of administrator groups rather than one single group leaves empty administrator groups as an option. In this case the group collapses – as a special case – just to ordinary, non-administrated groups.

Let's turn back now to our previous example. With this new definition of groups our example still works, without having to introduce separate groups for each individual subject.

Example 4 (Groups) *The group memory m is given by the term*

$$m = new_{memory};\ alloc\ g_1;\ alloc\ g_2;\ alloc\ g_3;$$

using the following abbreviations:

$$\begin{aligned} g_1 &= Admins\ [Ref\ 0];\ AddMember\ Tom,\ AddMember\ Harry \\ g_2 &= Admins\ [Ref\ 0];\ AddSubgroup\ (Ref\ 0);\ AddMember\ Peter \\ g_3 &= Admins\ [Ref\ 2];\ AddSubgroup\ (Ref\ 0);\ AddMember\ Jenny \end{aligned}$$

Again, we visualise this group structure by means of a figure (see Figure 3.6). In contrast with the former figures we have to deal with administrators and explicit members additionally to subgroups. We indicate administrator groups at the top, shallow members (i.e. members who are included directly in a group and not collected recursively in subgroups) in the middle and sub-groups at the bottom. Please note that group g_1 is administrating itself and group g_2. As a simple consequence we could postulate that only the members of group g_1 (i.e. currently Harry and Tom) were permitted to change the group structure of group g_2.

Once more, we add Dick to g_1. One might expect that we are done since we have already defined a function $AddMember : subject \to group \to group$, which is a result of the datatype definition of group, for adding members to groups. But we are only half way through, since we have to use function $lift : group\ Ref \to (group \to group) \to group\ memory \to group\ memory$ to lift the function from groups to group structures.

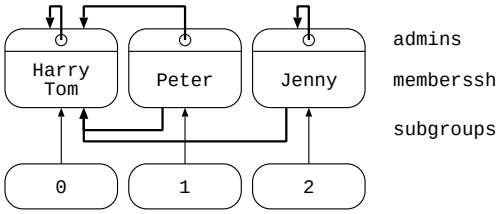


Figure 3.6: Group structure

Definition 14 (Adding subjects to groups)

$$addmember : subject \to group\,Ref \to group\,memory \to group\,memory$$
$$addmember\,p = lift\,(AddMember\,p)$$

Adding Dick to g_1 is then as simple as addmember Dick (Ref 0) m which is visualised in the following figure.

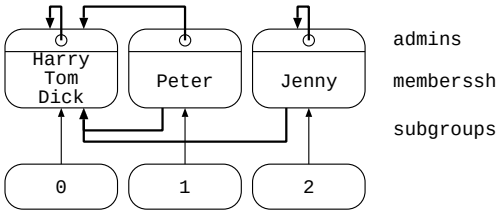


Figure 3.7: Group structure (Dick added to g_1)

Please note, that Dick has automatically become administrator both for group g_1 and g_2. □

It is often argued to lay responsibility with an individual subject rather than a group of subjects. The model also allows for single-user administrator groups, hence responsibility can be laid to a single subject – if required. Beyond that, our model is quite flexible since the administrator group is only referenced and not included literally.

Defining a function that allocates a new single-user group with given administrator group is straightforward.

Definition 15 (Single-user groups)

$$single\text{-}user_{group} : subject \to group\,Ref \to group\,mem \to group\,mem$$
$$single\text{-}user_{group}\,p\,r = alloc\,(AddMember\,p\,(Admins\,[r]))$$

Example 5 (Single-user groups) *As an example we add some single-user group consisting of Dick only and having self-referring administrator*

group to the group structure m which is depicted in Figure 3.6. This is achieved by the term single-user$_{group}$ Dick (freshref m) m and shown in Figure 3.8. □

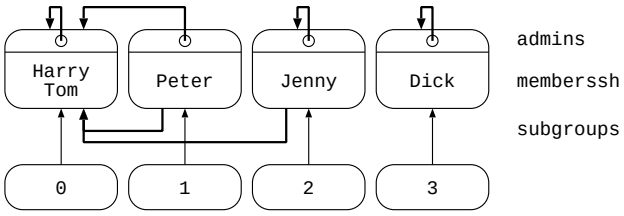


Figure 3.8: Group structure (single-user group "Dick" added)

Excursion: Groups are instances of $myreferences_{axclass}$

In footnote 3 we have mentioned that strictly speaking we had to show that type *group* is an instance of axiomatic type class $myreferences_{axclass}$. This is due to the fact that only those types τ *memory* are instances of type constructor class $references_{axclass}$ and thus may use the definitions and axioms of references freely where type τ is an instance of axiomatic type class $myreferences_{axclass}$. Again we have to define operations and prove their well-behaviour with respect to the propositions given in the axiomatic type class. First the operations[6].

Definition 16 (Types with references – operations) *The definitions of $myreferences_{remove}$ and $myreferences_{map}$ are just standard and therefore omitted.*

$$
\begin{array}{lll}
myreferences\ m\ (Admins\ []) & = \{\} \\
\mid \quad m\ (Admins\ (x :: xs)) & = (\textbf{if}\ x \in references\ m\ \textbf{then}\ \{x\}\ \textbf{else}\ \{\}\ ) \cup \\
 & \quad myreferences\ m\ (Admins\ xs) \\
\mid \quad m\ (AddMember\ p\ g) & = myreferences\ m\ g \\
\mid \quad m\ (AddSubgroup\ r\ g) & = \textbf{if}\ r \in references\ m \\
 & \qquad \textbf{then}\ \{r\} \cup myreferences\ m\ g \\
 & \qquad \textbf{else}\ myreferences\ m\ g \\
myreferences_{add}\ m\ r\ g = \textbf{if}\ r \in references\ m\ \textbf{then}\ AddSubgroup\ r\ g\ \textbf{else}\ g &
\end{array}
$$

The proofs are standard inductions.

Lemma 2 (Types with references – operations meet specification) *The operations on types containing references defined for type group meet specification $myreferences_{axclass}$. This is shown by an easy induction on the datatype groups.*

[6]We write $x :: xs$ for concatenation of element x with list xs

Once more we are in the position to declare an instance of an axiomatic type class.

Definition 17 (Instance of $myreferences_{axclass}$) *Type group is an instance of axiomatic type class $myreferences_{axclass}$.*

$$\textbf{\textit{instance}}\ group\ \textbf{\textit{of}}\ myreferences_{axclass}$$

You might have wondered how groups can have self-referring references (as e.g. the administrator group of g_1). In all of these cases the reference to the administrator group and the group itself coincide. At first sight this might suggest that our definitions are not well-founded. In the next section where we deal with membership we will see that self-references do not necessarily give rise to circular definitions.

3.1.2 Members

So far all of our definitions have been well-founded. Even the definition $\lambda\, m.\ alloc\,(Admins\,[freshref\ m])\ m$ is well-founded, though you might think at first sight it isn't. From a syntactic point of view (which is the only adequate point of view) the newly created object simply happens to contain a natural number. It's only in our mind that we interpret this (semantically) as being a self-reference. Problems arise, though, when trying to define recursive functions that make the "self-referring structure" explicit. A good candidate for such an ill-formed definition is "members". A straightforward attempt to define members fails for exactly this reason:

$$\begin{array}{l}
members : group\ memory \to group\ Ref \to subject\ list \\
members\ m\ r = members_{aux}\ m\ (dref\ r\ m) \\
members_{aux} : group\ memory \to group\ option \to subject\ list \\
\begin{array}{lll}
members_{aux}\ m\ None & = & \{\} \\
\quad | \quad m\ (Some\ (Admins\ [])) & = & \{\} \\
\quad | \quad m\ (Some\ (Admins\ (x :: xs))) & = & (\boxed{members_{aux}\ m\ (dref\ x\ m)}) \cup \\
 & & (members_{aux}\ m\ (Some\ (Admins\ xs))) \\
\quad | \quad m\ (Some\ (AddMember\ p\ g)) & = & \{p\} \cup (members_{aux}\ m\ (Some\ g)) \\
\quad | \quad m\ (Some\ (AddSubgroup\ r\ g)) & = & (\boxed{members_{aux}\ m\ (dref\ r\ m)}) \cup \\
 & & (members_{aux}\ m\ (Some\ g))
\end{array}
\end{array}$$

Evaluating the expression

$$members\ (alloc\ (Admins\ [freshref\ m])\ m)\ (freshref\ m)$$

for any memory m would loop forever as the following reduction demonstrates (with $x \rightsquigarrow_r y$ meaning "x reduces to y due to r" and m' abbreviating the term $alloc\,(Admins\,[freshref\ m])\ m$):

$$
\begin{array}{l}
members\ m'\ (freshref\ m) \rightsquigarrow_{definition\ of\ members} \\
members_{aux}\ m'\ (dref\ (freshref\ m)\ m') \rightsquigarrow_{definition\ 4} \\
members_{aux}\ m'\ (Some\ (Admins\ [freshref\ m])) \rightsquigarrow_{definition\ of\ members_{aux}} \\
members_{aux}\ m'\ (dref\ (freshref\ m)\ m')
\end{array}
$$

Of course, function $members_{aux}$ is not well-founded due to the recursive but not primitive recursive call $members_{aux}\ m\ (dref\ r\ m)$, which is highlighted by surrounding boxes. There are three ways to remedy this ill-formed definition. First avoiding the recursive call at all, second semantically guaranteeing that the structure of references is non-circular and third admitting circularities in memories, ignoring all circularities, though for functions dealing with memories.

Avoiding the recursive call at all seems more like a patch. Indeed, in most cases the result would be patchwork. In case of constructor *Admins*, though, it is reasonable to avoid the recursive call at all. Being member of the administrator group gives rise to some additional rights and thus is more kind of an administrative issue which does not necessarily imply membership in the group. If you really want to ensure that members of the administrator group are also members of the whole group, you are free to restrict group memories by some predicate that has to be invariant for all updates of the group memory.

Guaranteeing that the structure of references is non-circular is a *semantic* argument. We have pointed out several times already that semantic arguments are quite adequate in informal mathematical settings but are cumbersome in rigorous formalisations. This is true for this solution, too. Requiring memories to be non-circular requires carrying around the respective predicate and proving invariance of the predicate for any update since any change of the memory might violate this property. This way all formulas are getting inflated.

Once more the preferable solution in our rigorous formal setting is of *syntactic* nature. The idea behind this solution is similar to the intuition behind the definition of function $references_{obj}$ (see Definition 7). Group structures may be cyclic. The definition of membership takes care, though, that the algorithm refrains from calling the function recursively each time a cycle occurs. This way the definition gets well-founded even if the underlying group structure was not. Let's see now, how to turn these solutions in a concrete and well-founded definition of membership.

As we have said, we avoid the recursive call in case of constructor *Admins* at all by defining $members_{aux}\ m\ (Some\ (Admins\ xs)) = \{\}$. In case you would like to restrict group structures only to those where the administrators really are members of the respective group you have to ensure this property using a predicate.

Excursion: Administrators are members

To ensure that administrators of a group are members too, define a function *admins* which extracts the references to the administrator groups from a group.

Definition 18 (Administrator groups of a group)

$$
\begin{array}{l}
admins : group\ Ref \to group\ memory \to (group\ Ref)\ list \\
admins\ r\ m = admins_{aux}\ (dref\ r\ m) \\
admins_{aux} : group \to (group\ Ref)\ list \\
admins_{aux}\ (Admins\ xs) \quad\quad = xs \\
\quad | \quad\quad (AddMember\ p\ g) \ = admins_{aux}\ g \\
\quad | \quad\quad (AddSubgroup\ r\ g) = admins_{aux}\ g
\end{array}
$$

Next define a predicate *admins-are-members*, expressing that the administrators of a group are also members[7] of the group itself.

Definition 19 (Admins are members) *Relation admins-are-members decides for a particular group if all members of the administrator group are members of the group, too.*

$$
\begin{array}{l}
admins\text{-}are\text{-}members : group\ Ref \to group\ memory \to bool \\
admins\text{-}are\text{-}members\ r\ m = \\
\quad \forall p.\ p \in \bigcup_{x \in\ admins\ r\ m} members\ m\ x\ \Rightarrow\ p \in members\ m\ r
\end{array}
$$

The following lemma is a trivial consequence of this definition:

Lemma 3 (Self-referring administrator groups well-formed) *All groups where the administrator group and the group itself coincide satisfy predicate admins-are-members.*

$$\forall r\ m.\ admins\ r\ m = [r] \Rightarrow\ admins\text{-}are\text{-}members\ r\ m$$

This lemma, though quite trivial in nature, has a surprising consequence. It guarantees the existence of empty groups as well as single-user groups that are administrated and well-formed w.r.t predicate *admins-are-members*.

Lemma 4 (Empty groups well-formed) *There are empty administrated groups, i.e. groups with no members but non-empty set of administrator groups, that satisfy predicate admins-are-members.*

$$
\begin{array}{l}
\forall r\ m.\ dref\ r\ m = Admins\ [r] \Rightarrow members\ m\ r = \{\}\ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad admins\text{-}are\text{-}members\ r\ m
\end{array}
$$

[7] For the definition of *admins-are-members* we fall back upon Definition 21 of members which is deferred to page 50.

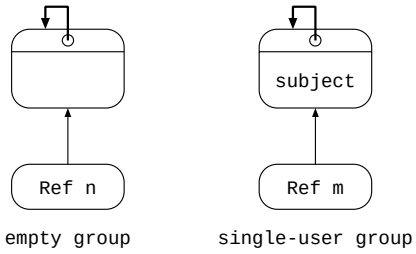


Figure 3.9: Empty groups and single-user groups

Lemma 5 (Single-user groups well-formed) *There are single-user groups, i.e. groups with exactly one member, that satisfy predicate admins-are-members.*

$$\forall r\ m\ p.\ dref\ r\ m = AddMember\ subject\ (Admins\ [r]) \Rightarrow \\ members\ m\ r = \{subject\}\ \wedge \\ admins\text{-}are\text{-}members\ r\ m$$

Please note that there are a number of groups containing zero or one member that do not satisfy the predicate *admins-are-members*. Well-formedness thus is only guaranteed for "canonical" groups.

Predicate *admins-are-members* seems to capture our intuition to include administrators into groups. Unfortunately, it is not structure persevering as far as the adding of subjects to groups is concerned. Figure 3.10 shows an counterexample.

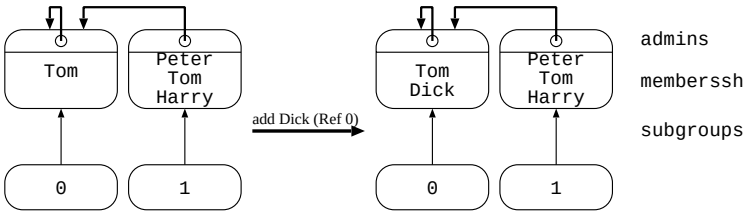


Figure 3.10: *admins-are-members* not structure preserving

There is a stronger predicate (in the sense that it admits less group structures) which is structure preserving in the just mentioned sense and still captures the essence of our intuition.

Definition 20 (Admins are subgroup) *Relation admins-are-subgroups decides for a particular group if the administrator group is also a subgroup*[8] *of the group.*

[8]For the definition of *admins-are-subgroups* we fall back upon the Definition 23 of subgroups which is deferred to page 54.

$$admins\text{-}are\text{-}subgroups : group\ Ref \to group\ memory \to bool$$
$$admins\text{-}are\text{-}subgroups\ r\ m = admins\ r\ m \subseteq subgroups\ m\ r$$

Both states in Figure 3.10 violate predicate *admins-are-subgroups* – hence the figure is no counterexample for *admins-are-subgroups*. Changing the example slightly by taking the left group to be subgroup of the right one, the transition preserves the structure. Apart from the concrete example we can prove that predicate *admins-are-subgroups* generally is structure preserving in the above mentioned sense.

Lemma 6 (*admins-are-subgroups* structure preserving)

$$admins\text{-}are\text{-}subgroups\ r\ m \Rightarrow admins\text{-}are\text{-}subgroups\ r\ (addmember\ p\ r\ m)$$

Finally we show that *admins-are-subgroups* really is more restrictive than *admins-are-members*.

Lemma 7 (*admins-are-subgroups* stronger than *admins-are-members*)

$$admins\text{-}are\text{-}subgroups\ r\ m \Rightarrow admins\text{-}are\text{-}members\ r\ m$$

After the detour on administrator groups let's turn back to a correct definition of members. Recall that we have eliminated the recursive function call for constructor *Admins* but the non-primitive-recursive call for constructor *AddSubgroup* still remains. To account for a well-founded definition of *members* we have to ignore all cycles in computations. In order to recognise when a cycle has been reached, all previously visited groups are stored in an auxiliary parameter.

Definition 21 (Members)

$$\begin{aligned}
&members : group\ memory \to group\ Ref \to subject\ list\\
&members\ m\ r = members_{aux}\ m\ \{r\}\ (dref\ r\ m)\\
&members_{aux} : group\ memory \to (group\ Ref)\ list \to group\ option \to subject\ list\\
&members_{aux}\ m\ v\ None = \{\}\\
&\quad |\quad m\ v\ (Some\ (Admins\ xs)) = \{\}\\
&\quad |\quad m\ v\ (Some\ (AddMember\ p\ g)) = \{p\} \cup (members_{aux}\ m\ v\ (Some\ g))\\
&\quad |\quad m\ v\ (Some\ (AddSubgroup\ r\ g)) =\\
&\qquad\qquad \textbf{\textit{if}}\ (r \in v) \vee (r \notin references\ m)\\
&\qquad\qquad\ \ \textbf{\textit{then}}\ members_{aux}\ m\ v\ (Some\ g)\\
&\qquad\qquad\ \ \textbf{\textit{else}}\ \ (members_{aux}\ m\ (v \cup \{r\})\ (dref\ r\ m)) \cup\\
&\qquad\qquad\qquad\qquad (members_{aux}\ m\ (v \cup \{r\})\ (Some\ g))
\end{aligned}$$

Before we turn to the technicalities of the well-formedness proof for this definition we shall illustrate the functionality of *members* by example (see Example 4 for the definition of "m").

Example 6 (Members)

$$\begin{array}{l} members\ m\ (Ref\ 0) = \{Tom, Harry\} \\ members\ m\ (Ref\ 1) = \{Tom, Harry, Peter\} \\ members\ m\ (Ref\ 2) = \{Tom, Harry, Jenny\} \end{array}$$

Now the technicalities. Even this definition of *members* is not primitive recursive, though it is certainly total. In order to achieve a well-formed definition we have to use a stronger definition principle: "total recursion".

Excursion: primitive and total recursion

We shall explain total recursion now and show that primitive recursion is just a special case of the more general total recursion. Let's recall primitive recursion first. *Primitive recursion* is based on the idea that each function call within the function body has to be structurally smaller than each function call in the head of the function. Take for example the inductive definition of lists over type α

$$\textbf{datatype}\ \alpha\ list = [] \mid Cons\ \alpha\ (\alpha\ list)$$

and of a primitive recursive function to concat two lists:

$$\begin{array}{l} concat : \alpha\ list \to \alpha\ list \to \alpha\ list \\ concat\ []\ ls \qquad\qquad\quad = ls \\ \underbrace{concat\ (Cons\ l\ ls)\ ls'}_{head} = \underbrace{Cons\ l\ (concat\ ls\ ls')}_{body} \end{array}$$

In the primitive recursive function call $concat\ ls\ ls'$ in the body the argument ls is structurally smaller than the corresponding argument $Cons\ l\ ls$ in the function head $concat\ (Cons\ l\ ls)\ ls'$ in the sense that one occurrence of constructor *Cons* has been peeled off. Given that any term is finite and thus may only be composed of finitely many constructors, the process of peeling off constructors by recursive function calls eventually must terminate rendering the definition of *concat* well-founded. Please note that this argument is once more purely *syntactic*. That's why it can be recognised automatically by a machine.

In contrast, *total recursion* is based on a *semantic* argument – hence it is more general but not as schematic as primitive recursion. The user has to give a semantic termination proof – requiring intuition, which a machine does not posses. To account for a complete definition of a function the programmer has to accompany the definition with a *measure function* that maps the arguments of the function to an ordered set (to be precise, the ordering has to be Noetherian, i.e. it must not contain any infinitely descending chain). The requirement is that in each defining equation the measure of the arguments in the function head has to be strictly greater than

the measure for any occurrence of the function in the body of the equation. For our example of concatenating lists the measure function is as trivial as:

$$measure_{concat} : \alpha\ list \rightarrow nat$$
$$measure_{concat} = length$$

Function *length* counts the number of occurrences of constructor *Cons* in a term. Each time constructor *Cons* is peeled off, the length is diminished.

In general, a measure function can be computed for each inductive datatype such that the measure strictly decreases in all primitive recursive function definitions (and the proof for this proposition can be constructed uniformly). This way, primitive recursion is always well-founded and can be seen as a special case of total recursion. □

After this detour we return to the definition of the function $members_{aux}$ and show that above Definition 21 is an instance of total recursion. To do so, we have to define a measure function first and then to show that the measure decreases for every recursive function call. The function definition contains the following recursive function calls in the head and body respectively:

Head	*Body*
$members_{aux}\ m\ v\ (Some\ (AddMember\ p\ g))$	$members_{aux}\ m\ v\ (Some\ g)$
$members_{aux}\ m\ v\ (Some\ (AddSubgroup\ r\ g))$	$members_{aux}\ m\ v\ (Some\ g)$
$members_{aux}\ m\ v\ (Some\ (AddSubgroup\ r\ g))$	$members_{aux}\ m\ (v \cup \{r\})\ (dref\ r\ m)$
$members_{aux}\ m\ v\ (Some\ (AddSubgroup\ r\ g))$	$members_{aux}\ m\ (v \cup \{r\})\ (Some\ g)$

Looking at the recursive function calls more closely you notice two different kinds of calls. The recursive calls in the first two lines are in principle primitive recursive (except for the nesting within datatype *option*). The recursive calls in the third and fourth line are not at all primitive recursive and require an explicit termination argument. In these cases termination is guaranteed by the auxiliary first argument which logs the set of all previously visited references. Subtracting this set from the set of all valid references results in a set which shrinks as $members_{aux}$ goes by. This process might not terminate if the set of references was infinite. At this point we realize why it was reasonable to represent references in memories by lists rather than sets (see discussion in Section 2.3). Lists are finite by definition and hence do not require any additional semantic argument (which is quite cumbersome in a rigorously formal setting).

In parallel with the two different kinds of function calls the measure function has to provide two ways of decreasing its value (depending on the respective function call). These two alternatives have to be combined in such a way that they do not interfere improperly. The second alternative, which deals with the set of all previously visited references, is dominating, whereas the first one only plays a secondary role. This kind of dependency reminds of lexicographical orderings where the leading positions are dominating over the subsequent ones.

Indeed, this intuition immediately leads to a suitable Noetherian ordering. We use a binary lexicographical ordering (i.e. each word consists of exactly two letters) based on natural numbers:

$$\begin{array}{l} < : nat \times nat \to bool \\ (n,\, m) < (n',\, m') \iff n < n' \vee (n = n' \wedge m < m') \end{array}$$

Strictly speaking, measure functions do only map exactly one argument of the function under consideration to an ordered set. We weaken this requirement allowing measure functions to map several arguments to the ordered set. From a theoretical point of view this "weakening" is unproblematic, since we could combine these multiple arguments into a single argument using product types.

Definition 22 (Measure function for $members_{aux}$)

$$\begin{array}{lll} measure : group\ memory \to (group\ Ref)\ list \to group\ option \to nat \times nat & & \\ measure\ m\ v\ None & = (|\, references\ m - v\, |,\ 0) & \\ \mid \quad m\ v\ (Some\ (Admins\ xs)) & = (|\, references\ m - v\, |,\ 0) & \\ \mid \quad m\ v\ (Some\ (AddMember\ p\ g)) & = (measure\ m\ v\ (Some\ g)) + (0,\ 1) & \\ \mid \quad m\ v\ (Some\ (AddSubgroup\ r\ g)) & = (measure\ m\ v\ (Some\ g)) + (0,\ 1) & \end{array}$$

It is quite obvious that the measure function stores the number of all yet unvisited references in its first component.

Lemma 8 (Measure for $members_{aux}$ constant in first component) *The first component of measure computes the cardinality of the set of all yet unvisited references.*

$$(measure\ m\ v\ x).1 = |\, references\ m - v\, |$$

Proof: Nested induction on *option* (first induction) and *group* (second induction).

This auxiliary lemma is required for the following lemma which shows that any increase in the set of already visited references strictly decreases the first component of the measure function and hence also the whole measure function.

Lemma 9 (Visited references decrease *measure*) *Let $r \notin v$ and $r \in references\ m$.*

$$(measure\ m\ v\ x).1 > (measure\ m\ (v \cup \{r\})\ y).1$$

Proof: Lemma 8 transforms the proposition to $|\, references\ m - v\, | > |\, references\ m - (v \cup \{r\})\, |$. This is trivial due to set theoretic arguments using the assumptions.

Lemma 10 (Measure for $members_{aux}$ strictly decreases) *For all recursive function calls of function $members_{aux}$ the measure function strictly decreases. For Cases (3) and (4) the propositions only hold provided that $r \notin v$ and $r \in references\, m$.*

$$\begin{array}{lcll} measure\ m\ v\ (Some\ (AddMember\ p\ g)) & > & measure\ m\ v\ (Some\ g) & (1)\\ measure\ m\ v\ (Some\ (AddSubgroup\ r\ g)) & > & measure\ m\ v\ (Some\ g) & (2)\\ measure\ m\ v\ (Some\ (AddSubgroup\ r\ g)) & > & measure\ m\ (v \cup \{r\})\ (dref\ r\ m) & (3)\\ measure\ m\ v\ (Some\ (AddSubgroup\ r\ g)) & > & measure\ m\ (v \cup \{r\})\ (Some\ g) & (4) \end{array}$$

Proof: The proofs of Propositions (1) and (2) are trivial by definition of *measure*. For the proofs of Propositions (3) and (4) it suffices to show – by definition of the ordering – that the corresponding proposition holds for the first components. Lemma 9 completes the proofs. The side conditions of this lemma are respected since the corresponding recursive function calls of $members_{aux}$ are guarded by the expression "$(r \in v) \vee (r \notin references\, m)$" and are only executed if the guard is negated.

This lemma together with the property that each lexicographic ordering is Noetherian ensures the well-formedness of Definition 21.

3.1.3 Subgroups

Function *subgroups*, which determines the subgroups of a given group, is quite analogous to function *members*. The proof of well-formedness is based on the same measure function *measure* and is conducted in complete analogy. Again, ill-formed references are simply ignored.

Definition 23 (Subgroups)

$$\begin{array}{l} subgroups : group\ memory \to group\ Ref \to (group\ Ref)\ list\\ subgroups\ m\ r = \{r\} \cup (subgroups_{aux}\ m\ \{r\}\ (dref\ r\ m))\\ subgroups_{aux} :\\ \quad group\ memory \to (group\ Ref)\ list \to group\ option \to (group\ Ref)\ list\\ subgroups_{aux}\ m\ v\ None = \{\}\\ \quad |\ m\ v\ (Some\ (Admins\ xs)) = \{\}\\ \quad |\ m\ v\ (Some\ (AddMember\ p\ g)) = subgroups_{aux}\ m\ v\ (Some\ g)\\ \quad |\ m\ v\ (Some\ (AddSubgroup\ r\ g)) =\\ \qquad \textbf{\textit{if}}\ (r \in v) \vee (r \notin references\, m)\\ \qquad\ \ \textbf{\textit{then}}\ subgroups_{aux}\ m\ v\ (Some\ g)\\ \qquad\ \ \textbf{\textit{else}}\ \{r\} \cup\\ \qquad\qquad (subgroups_{aux}\ m\ (v \cup \{r\})\ (dref\ r\ m)) \cup\\ \qquad\qquad (subgroups_{aux}\ m\ (v \cup \{r\})\ (Some\ g)) \end{array}$$

Again, we shall illustrate the functionality of the function first:

Example 7 (Subgroups)

$$\begin{aligned} &subgroups\ m\ (Ref\ 0) = \{Ref\ 0\} \\ &subgroups\ m\ (Ref\ 1) = \{Ref\ 1, Ref\ 0\} \\ &subgroups\ m\ (Ref\ 2) = \{Ref\ 2, Ref\ 0\} \end{aligned}$$

Please note that function *subgroups* is reflexive, i.e. $r \in subgroups\ m\ r$.

The syntactic analogy between functions *members* and *subgroups* suggests that there is a correspondence between these functions. Indeed this correspondence can be captured formally by a simple theorem (simple to be expressed but not quite as easy to be proved).

Primitive recursive function $members_{sh}$ plays a vital role in the lemma. The function computes the set of all shallow members (see page 43 where we have called all members that are not due to subgroups "shallow members") of a group.

$$\begin{aligned} &members_{sh} : group\ memory \to group\ Ref \to subject\ list \\ &members_{sh}\ m\ r = members_{sh_{aux}}\ (dref\ r\ m) \\ &\\ &members_{sh_{aux}} : group\ option \to subject\ list \\ &members_{sh_{aux}}\ None \quad\ = \{\} \\ &\quad | \qquad (Some\ g) = members_{sh_{aux'}}\ g \\ &\\ &members_{sh_{aux'}} : group \to subject\ list \\ &members_{sh_{aux'}}\ (Admins\ xs) \qquad = \{\} \\ &\quad | \qquad (AddMember\ p\ g) \ = \{p\} \cup (members_{sh_{aux'}}\ g) \\ &\quad | \qquad (AddSubgroup\ r\ g) = members_{sh_{aux'}}\ g \end{aligned}$$

This function is kind of a mediator between the functions *members* and *subgroups*: the members of a group can be computed by collecting the shallow members of all subgroups.

$$members\ m\ r = \bigcup_{r' \in subgroups\ m\ r} members_{sh}\ m\ r'$$

The proof of this proposition is not immediate because the principle of structural induction does not suffice for this purpose. But which induction principle is sufficiently expressive, then? Each recursive definition scheme comes together with a corresponding induction scheme. Structural induction, the most famous induction scheme, corresponds to primitive recursion, which undoubtedly is the most famous definition scheme. In the definition of *members* and *subgroups* we have used total (or sometimes also called well-founded) recursion. The corresponding induction principle is called *well-founded induction*. Just as primitive recursion turned out to be a special case of total recursion, structural induction is a special case of the more general well-founded induction.

Excursion: structural and well-founded induction

Let's recall structural induction. Assume you would like to prove a proposition $P\ n$ for all n that are elements of a previously defined inductive datatype N. To establish $\forall n.\ P\ n$ it suffices to show that P is an invariant for all constructors of N. Take for example inductive datatype *list* with constructors $[]$ and *Cons*. In order to prove a proposition $\forall ls : list.\ P\ ls$ it suffices to prove that P holds for the constant constructor $[]$ and that *Cons* preserves P.

$$(P\ [] \wedge (\forall l\ ls.\ P\ ls \Rightarrow P\ (Cons\ l\ ls))) \Rightarrow \forall ls.\ P\ ls$$

In the induction step you may assume that the same proposition holds for terms with one occurrence of the constructor stripped off. Obviously, induction still would work if in the induction step more than one constructor was stripped off. Well-founded induction takes up this point.

Definition 24 (Well-founded induction) *Let $P : \alpha \to \alpha$ be an arbitrary predicate and $< : \alpha \to \alpha \to bool$ a Noetherian relation.*

$$(\forall x < y.\ P\ x \Rightarrow P\ y) \Rightarrow \forall x.\ P\ x$$

You may read this as follows: To prove $P\ y$ in the induction step you may assume that proposition P holds for all $x < y$. Please note that this induction principle can do without explicit induction basis. □

On our way to the desired theorem we need one further (simple) lemma which expresses that all shallow members are also proper members.

Lemma 11 ($members_{sh}$ subset $members_{aux}$)

$$members_{sh_{aux'}}\ g \subseteq members_{aux}\ m\ v\ (Some\ g) \qquad (1)$$
$$members_{sh_{aux}}\ g_{opt} \subseteq members_{aux}\ m\ v\ g_{opt} \qquad (2)$$

Proof: Proposition (1): induction on g; proposition (2): nested induction – first induction on *option* and second on *group*

Finally, we prove above mentioned theorem which characterises the relationship between *members* and *subgroups*. Please note that *subgroups* returns lists rather than sets and hence this equation could be taken as definitional equation for *members* (using list comprehension rather than set comprehension).

Theorem 1 (Alternative definition of *members*)

$$members\ m\ r = \bigcup_{r' \in subgroups\ m\ r} members_{sh}\ m\ r'$$

Proof: Taking advantage of the set theoretic equation $\bigcup_{r' \in \{r\} \cup S} f\ r' = (f\ r) \cup \bigcup_{r' \in S} f\ r'$ this theorem is a special case of Lemma 12 instantiating $g_{opt} \rightsquigarrow dref\ r\ m$ and $v \rightsquigarrow \{r\}$.

As we have pointed out earlier, we have to use well-founded induction instead of structural induction for the correctness proof of Lemma 12. Induction is performed on triples of type $group\ memory \times (group\ Ref)\,list \times group\ option$. The required ordering is defined on the basis of *measure*.

$$\begin{aligned}
&< \ :\ (group\ memory \times (group\ Ref)\,list \times group\ option) \times \\
&\qquad (group\ memory \times (group\ Ref)\,list \times group\ option) \to bool \\
&(m,\ v,\ g_{opt}) < (m',\ v',\ g_{opt}{}') \iff (measure\ m\ v\ g_{opt}) < (measure\ m'\ v'\ g_{opt}{}')
\end{aligned}$$

The theorem could not be proved directly by induction. The proposition has to be strengthened first to be a valid invariant in the induction steps of the proof. Lemma 12 proves a strengthened version of the theorem.

Lemma 12 (Alternative definition of *members* – strengthening)

$$\begin{aligned}
&members_{aux}\ m\ v\ g_{opt} = \\
&(\bigcup_{r' \in subgroups_{aux}\ m\ v\ g_{opt}} members_{sh_{aux}}\ (dref\ r'\ m)) \cup (members_{sh_{aux}}\ g_{opt})
\end{aligned}$$

Proof: First well-founded induction on $(m,\ v,\ g_{opt})$ and then case distinction on g_{opt}. Let m, v, g_{opt} be arbitrary and assume the proposition being true for all m', v', $g_{opt'}$ strictly smaller than m, v, g_{opt}.

Cases $g_{opt} = None$ and $g_{opt} = Some\,(Admins\,xs)$: trivial

Case $g_{opt} = Some\,(AddMember\,p\,g)$:
In this case we have to prove the following equation:

$$\begin{aligned}
&members_{aux}\ m\ v\ (Some\ (AddMember\,p\,g)) = \\
&(\bigcup_{r' \in subgroups_{aux}\ m\ v\ (Some\,(AddMember\,p\,g))} members_{sh_{aux}}\ (dref\ r'\ m)) \cup \\
&(members_{sh_{aux}}\ (Some\ (AddMember\,p\,g)))
\end{aligned}$$

Expanding the definitions of $members_{aux}$, $subgroups_{aux}$ and $members_{sh_{aux}}$ this equation rewrites to:

$$\begin{aligned}
&\{p\} \cup (members_{aux}\ m\ v\ (Some\ g)) = \\
&(\bigcup_{r' \in subgroups_{aux}\ m\ v\ (Some\ g)} members_{sh_{aux}}\ (dref\ r'\ m)) \cup \\
&\{p\} \cup (members_{sh_{aux'}}\ g)
\end{aligned}$$

Due to Lemma 10 Proposition (1) we may apply the induction hypothesis resulting in the following subgoal and hence we are done for this case.

$$\begin{aligned}
&\{p\} \cup (members_{aux}\ m\ v\ (Some\ g)) = \\
&(members_{aux}\ m\ v\ (Some\ g)) \cup \{p\}
\end{aligned}$$

Case $g_{opt} = Some\,(AddSubgroup\,r\,g)$:
We have to prove the following equation:

$$\begin{aligned}
&members_{aux}\ m\ v\ (Some\ (AddSubgroup\,r\,g)) = \\
&(\bigcup_{r' \in subgroups_{aux}\ m\ v\ (Some\,(AddSubgroup\,r\,g))} members_{sh_{aux}}\ (dref\ r'\ m)) \cup \\
&(members_{sh_{aux}}\ (Some\ (AddSubgroup\,r\,g)))
\end{aligned}$$

Expanding the definitions of $members_{aux}$, $subgroups_{aux}$ and $members_{sh_{aux}}$, this equation rewrites to:

$$
\begin{array}{l}
\textbf{if } (r \in v) \vee (r \notin references\ m) \\
\quad \textbf{then } members_{aux}\ m\ v\ (Some\ g) \\
\quad \textbf{else } (members_{aux}\ m\ (v \cup \{r\})\ (dref\ r\ m)) \cup \\
\qquad\quad (members_{aux}\ m\ (v \cup \{r\})\ (Some\ g)) = \\
(\bigcup_{r' \in S} members_{sh_{aux}}\ (dref\ r'\ m)) \cup (members_{sh_{aux'}}\ g) \\
with \quad S = \textbf{if } (r \in v) \vee (r \notin references\ m) \\
\qquad\qquad\quad \textbf{then } subgroups_{aux}\ m\ v\ (Some\ g) \\
\qquad\qquad\quad \textbf{else } \{r\} \cup \\
\qquad\qquad\qquad\quad (subgroups_{aux}\ m\ (v \cup \{r\})\ (dref\ r\ m)) \cup \\
\qquad\qquad\qquad\quad (subgroups_{aux}\ m\ (v \cup \{r\})\ (Some\ g))
\end{array}
$$

This horrendous term simplifies when performing case distinction on r. Assume $(r \in v) \vee (r \notin references\ m)$ – we then have to prove the following goal.

$$
\begin{array}{l}
members_{aux}\ m\ v\ (Some\ g) = \\
(\bigcup_{r' \in subgroups_{aux}\ m\ v\ (Some\ g)} members_{sh_{aux}}\ (dref\ r'\ m)) \cup (members_{sh_{aux'}}\ g)
\end{array}
$$

Similar to case "$Some\ (AddMember\ p\ g)$" we may apply induction hypothesis due to Lemma 10 Proposition (2) and then we are done for this path of the current case. Assume now $\neg((r \in v) \vee (r \notin references\ m))$ which equals $(r \notin v) \wedge (r \in references\ m)$ and leads to the following subgoal

$$
\begin{array}{l}
(members_{aux}\ m\ (v \cup \{r\})\ (dref\ r\ m)) \cup (members_{aux}\ m\ (v \cup \{r\})\ (Some\ g)) = \\
(\bigcup_{r' \in S} members_{sh_{aux}}\ (dref\ r'\ m)) \cup (members_{sh_{aux'}}\ g) \\
with \quad S = \{r\} \cup \\
\qquad\quad (subgroups_{aux}\ m\ (v \cup \{r\})\ (dref\ r\ m)) \cup \\
\qquad\quad (subgroups_{aux}\ m\ (v \cup \{r\})\ (Some\ g))
\end{array}
$$

Splitting the indexed union along the index set results in

$$
\begin{array}{l}
(members_{aux}\ m\ (v \cup \{r\})\ (dref\ r\ m)) \cup (members_{aux}\ m\ (v \cup \{r\})\ (Some\ g)) = \\
(members_{sh_{aux}}\ (dref\ r\ m)) \cup \\
(\bigcup_{r' \in subgroups_{aux}\ m\ (v \cup \{r\})\ (dref\ r\ m)} members_{sh_{aux}}\ (dref\ r'\ m)) \cup \\
(\bigcup_{r' \in subgroups_{aux}\ m\ (v \cup \{r\})\ (Some\ g)} members_{sh_{aux}}\ (dref\ r'\ m)) \cup \\
(members_{sh_{aux'}}\ g)
\end{array}
$$

In order to apply Lemma 10 Propositions (3) and (4) we have to ensure that $r \notin v$ and $r \in references\ m$. These conditions, though, are guaranteed by above assumption. Applying the induction hypothesis twice resolves the path of the current case and also finishes the proof of the whole lemma.

Q.E.D.

You might have wondered why we have not taken the theorem as definition of function *members*. The theorem gives us some kind of certainty that our

definitions were right. Indeed, we had to improve our previous definitions (the preliminary definitions are not documented in this paper) twice in order to get the theorem through: in the definition of *subgroups* the starting reference r had been forgotten and in the case distinction one recursive call of the function had been missed.

After all these technicalities let's stand back and see what we have achieved so far. We have defined an inductive datatype of groups which may be used as internal representation for organisational structures. Furthermore we have implemented standard operations to compute the members of a group and to establish the set of subgroups. All of these operations take a fixed group structure for granted. In reality, though, group structures are subject to permanent changes. There are a number of questions to be raised for changing group structures:

- How is update on groups represented?
- Which kinds of changes are permitted?
- Who may perform changes?
- How to prevent inadmissible changes?
- How to prevent un-authorised access?

We could answer all of these questions right away, but we postpone the answers to these questions for a moment. We will see later that changing group structures are just a special case of ordinary databases (taking groups as data).

3.2 Databases

Databases represent particular communication threads that are performed collaboratively by a multitude of users. Each database consists of a collection of data that is shared among the users. Just as for the changing group structures there are a number of (analogous) questions to be raised.

1. How is data represented?
2. How is update on data represented?
3. Which kinds of changes are permitted?
4. Who may perform changes?
5. How to prevent inadmissible changes?

6. How to prevent un-authorised access?

The first question is answered easily. Logos does not rely on any particular data structure. In powerful groupware systems one could even imagine higher-order structures (like higher-order functions) as data. Nevertheless, in most cases data will simply be tuples of values.

The answer to the second question is quite straightforward in a functional setting like ours. Just as in any other functional model, update on states has to be represented as functions mapping states to states. In our case, databases are characterised by the current state of the data space and by functions (operations) updating and accessing the data spaces (later we will see, that databases also map ACLs to updated ACLs).

The third question is a bit more subtle. Since we are building a general framework for authorisation in groupware systems, it is not our intention to anticipate concrete implementations. The possible changes are either determined by the groupware system itself or by the implementors of the database. For our purpose it is irrelevant who (groupware system or database implementor) provides the functionality. We simply assume that there is some instance which provides the underlying implementation.

The fourth question is at the heart of our model since authorisation is our major issue. The answer is a two layered approach. First layer is in the ACL. In the ACL the database manager assigns subjects to roles. If a subject is excluded from a role it can not have the respective access rights. But even if a subject is assigned some role this does not necessarily have any impact on the database. In order to be relevant, the implementation of the database must provide operations that are enabled for this role and disabled for others.

The fifth and sixth question are somewhat related. Both questions assume the existence of a set of operations (for update and access) and they raise the questions how to guarantee that these operations are the only way to update and access the state. The solution to this problem is to add a so-called "closed world assumption" which is a standard "trick" in theoretical computer science. Formally, closed world assumptions can be achieved by inductive definitions. Induction principles exactly capture the idea of closed worlds. Indeed, we will show later how to answer the fifth and sixth question using inductive sets.

In the sequel we will answer these six questions in greater detail. We will see that the answers do not anticipate any concrete groupware system and hence Logos truly deserves the denomination "framework".

3.2.1 Data space

The data space is the place where all the data is collected. Speaking in the terminology of typed functional languages, the data space has to be

polymorphic in the type of data in order to be independent of any particular type of data. This does not imply, though, that data spaces do not possess any uniform structure. On the contrary, all data spaces have the same structure: memories polymorphic in the type of data.

Definition 25 (Type of data spaces) *Assume any database d with data of type $data^d$. The corresponding data space possesses type $data^d$ memory.*

For sake of illustration we develop a running example which will be our companion over the next sections.

Example 8 (Discussion bboards – Data space) *The example deals with discussion bboards. Discussion bboards are a commonly accepted application of groupware systems. They allow for postings of notes and responses to these notes. Though postings and responses are similar in nature, they are different in content. Responses comprise a link to the original posting which the original posting does not. Furthermore the original posting determines the category it belongs to. The reply does not have this choice – it is assigned exactly the same categories the original document was assigned to.*

In order to define a data space for discussion bboards, we have to define the data (postings and responses) first. Original postings are tuples consisting of the following fields: author, date, categories and note. Responses do not contain field categories, but field parent, instead. This informal specification directly translates into record declarations.

record *main* =	***record*** *response* =
author : *subject*	*parent* : *main Ref*
date : *date*	*author* : *subject*
categories : *string list*	*date* : *date*
note : *string*	*note* : *string*

Given the type of data, it is easy to define the data space for discussion bboards.

$$data\text{-}space^{bboard} = data^{bboard}\ memory \quad with$$
$$data^{bboard} = \{|\ main : main,\ response : response\ |\}$$

□

3.2.2 Profile Documents

Amongst the many documents of a database one kind of documents is special: the profile documents. Profile documents collect user-specific or database-specific values. Data stored in the profile documents can be accessed particularly easily from other parts of the database. In spite of this special treatment of profile documents, they are just like any other document subject to access control.

From what we have said about profile documents you can see that there is basically no restriction on how profile documents look like. Since we are in a typed environment the only restriction is that profile documents are typed.

Definition 26 (Profile documents) *Assume d being the name of a database.*

$$\textbf{\textit{type}}\ profile^{d}$$

Example 9 (Discussion bboards – Profile documents) *For our example of discussion bboards we have only one database-specific document and a row of user-specific documents. The database-specific document contains a list of categories – the set of categories users may select from when editing documents. For each user we provide a user-specific document which determines whether the user wants to be notified*[9] *by eMail once a new document has be stored in the database. Since we are working in a functional environment, we may simply represent the set of user-specific documents as a function.*

$$\textbf{\textit{type}}\ profile^{bboard} = \{categories : string\,list,\ notification : subject \rightarrow bool\}$$

□

3.2.3 Roles

Roles are a means to distinguish different responsibilities locally within a particular database. Technically speaking, roles are no more than sets of distinguishable entities. Formally, we define for each database a separate type of roles.

Definition 27 (Roles) *Assume d being the name of any database. The type of roles for this database is given by the following type declaration.*

$$\textbf{\textit{type}}\ role^{d}$$

You may interject that by this definition roles are merely syntactic entities without meaning. The definition of roles nevertheless is meaningful because access control is spread over two levels of the Logos framework. The first level is the so-called *access control list (ACL)* of a database. In the ACL the database-manager assigns subjects to roles. If a subject is excluded from a role it cannot have the respective access rights. The second level concerns the operations of the database: Even if a subject is assigned a role this does not necessarily have any consequence for the database. In order to be

[9] We will not treat eMail notification in the remainder of the chapter any further. Notification only serves to demonstrate user-specific profile documents at this point.

relevant, the implementation of the database must provide operations that are enabled for this role and disabled for others. We will deal with ACLs and operations explicitly in the next sections.

Example 10 (Discussion bboards – Roles) *For the particular example of bboards we have chosen the following four roles:*

- **Manager:** *Managers have the most far-reaching rights of all roles. In particular, they may change the ACL (and only managers may change the ACL) and perform all operations that editors, authors or readers are permitted to.*
- **Author:** *Authors may post new documents (original postings just as responses). They are permitted to edit and delete their own documents. Furthermore they may read all documents.*
- **Editor:** *Editors are like authors with the difference that they may edit or delete any document - not just their own documents.*
- **Reader:** *Readers may read any document. Beyond that they have no additional rights.*

Defining the type of roles is canonical:

$$\textbf{\textit{datatype}}\ role^{bboard} = Manager \mid Editor \mid Author \mid Reader \qquad \square$$

In the literature there are a number of meanings assigned to the notions of groups, teams and roles – sometimes some of the notions even coincide. In our framework Logos we do differentiate between roles, teams and groups – their difference is subtle, though. As we have said, groups are a means to structure the overall set of subjects into entities. Some of these entities are assigned rights others serve only as interim entities to support the structuring. We will call those groups that have rights on their own also *teams*.

Since we present a rigorous formal model, we will be able to define all of theses notions precisely (in a mathematical sense). Regardless whether our definitions will get commonly accepted or not, our rigorous formal model can serve as a basis for a concise and hence fertile discussion of theses issues.

3.2.4 Access control list

The access control list (ACL) is the place where subjects are assigned roles. We have pointed out already that even if a subject is assigned a role this does not necessarily have any consequence for the database. In order to be relevant, the implementation of the database must provide operations that are enabled for this role and disabled for others.

One of our fundamentals is that we deal with groups rather than individual subjects. This decision is a strict generalisation in that individuals may always be represented as single-subject groups. This idea together with the already presented definition of roles (see Section 3.2.3) almost immediately leads to a canonical representation of ACLs in Logos. ACLs simply are functions mapping roles (names of roles) to sets of groups. Since we are only interested in finite sets of groups we choose lists of groups, instead.

Just as we have admitted hierarchic group structures in the NAB, we permit hierarchic role structures locally within databases. In order to account for changing group members we map roles to lists of group references rather than lists of groups themselves.

Definition 28 (ACL) *Assume d being the name of a database. The type of the ACL is given as $(role^d\ ACL)$ with type constructor ACL.*

$$\begin{array}{l}
\textbf{\textit{type}}\ \ \alpha\ ACL\text{-}groups = \alpha \to (group\ Ref)\ list \\
\textbf{\textit{type}}\ \ \alpha\ ACL\text{-}roles = \alpha \to \alpha\ list \\
\textbf{\textit{type}}\ \ \alpha\ ACL = \{groups : \alpha\ ACL\text{-}groups,\ roles : \alpha\ ACL\text{-}roles\}
\end{array}$$

As we have said, ACLs map roles to both lists of group references and lists of roles. The lists of roles represent dependencies between roles. For example they express that all managers are also readers of a database.

Example 11 (Discussion bboards – ACL) *There are a number of dependencies between the roles of the discussion bboards which are expressed by function $acl\text{-}roles_{bboard}$.*

$$\begin{array}{lll}
acl\text{-}roles^{bboard} & : role^{bboard}\ ACL\text{-}roles & \\
acl\text{-}roles^{bboard}\ Manager & = [] & \\
\mid \qquad Editor & = [Manager] & \\
\mid \qquad Author & = [Editor] & \\
\mid \qquad Reader & = [Author] &
\end{array}$$

Managers are maximal in the ordering of roles in the sense that no other role implies membership in the manager role (first case). The second case expresses that all managers also have editor-rights. In the third case we assign both managers and editors author rights. The last case, finally, states that members of all roles may read any document i.e. readers are minimal in the hierarchy of roles.

Managers may change the ACL while execution. For our example we therefore can only give a sample configuration of the ACL. Since this configuration is reached in two steps from the initial configuration we annotate the name with double "apostrophe".

$$\begin{array}{lll}
acl\text{-}groups^{bboard''} & : role^{bboard}\ ACL\text{-}groups & \\
acl\text{-}groups^{bboard''}\ Manager & = [Ref\ 0] & \\
\mid \qquad Editor & = [Ref\ 1] & \\
\mid \qquad Author & = [Ref\ 2] & \\
\mid \qquad Reader & = [] &
\end{array}$$

Given acl-roles^{bboard} and acl-groups$^{bboard}{''}$, it is easy to define the value of the corresponding ACL called $ACL^{bboard}{''}$.

$$\begin{aligned} ACL^{bboard}{''} &: role^{bboard}\, ACL \\ ACL^{bboard}{''} &= \{groups = acl\text{-}groups^{bboard}{''}, \\ &\qquad roles = acl\text{-}roles^{bboard}\} \end{aligned}$$

□

The dependencies between roles are established by the implementors of a database and are not subject to changes during execution. The lists of groups, in contrast, may be updated while execution. This distinction is stressed by the following function $lift_{ACL}$ which allows for updates in the lists of group references while leaving the lists of roles untouched. We will always use this function to achieve updates on ACLs.

Definition 29 (Group update in ACLs)

$$\begin{aligned} &lift_{ACL} : ((group\ Ref)\ list \to (group\ Ref)\ list) \to \alpha\ ACL \to \alpha\ ACL \\ &lift_{ACL}\ f\ acl = \{groups = \lambda r.\ f\ (acl.groups\ r),\ roles = acl.roles\} \end{aligned}$$

ACLs themselves are subject to access control, which in turn is determined by the ACL itself. There is kind of a circularity here which has a fundamental consequence. In order to be able to account for changes in the ACL, there must always be at least one subject having the right to change the settings in the ACL. Otherwise the ACL is stuck forever. This property is an invariant that has to be ensured by any system's implementation. In real systems this subject (or non-empty group of subjects) often is called manager of the database.

In order to determine if a subject may perform some operation we need a function *assignees* that computes all subjects assigned a particular role in the ACL. After a short detour we define function *assignees* formally.

Excursion: reflexive and transitive closure

As an auxiliary function we define *transclosure* which computes the reflexive and transitive closure of a relation. We use a non-standard representation of relations representing binary relations over type α as functions of type $\alpha \to \alpha\ list$. Assume f being a function of type $\alpha \to \alpha\ list$ and x_1 and x_2 being two elements of type α then x_1 and x_2 are related by f if and only if x_2 is in the set $f\ x_1$.

Definition 30 (Reflexive and transitive closure)

$$
\begin{array}{l}
transclosure : (\alpha \to \alpha\ list) \to (\alpha \to \alpha\ list) \\
transclosure = transclosure_{aux}\ \{\} \\
transclosure_{aux} : \alpha\ list \to (\alpha \to \alpha\ list) \to (\alpha \to \alpha\ list) \\
transclosure_{aux}\ v\ f\ r = \{r\} \cup \bigcup_{x \in f\ r}\ \textbf{\textit{if}}\ x \in v\ \textbf{\textit{then}}\ \{\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{\textit{else}}\ transclosure_{aux}\ (\{x\} \cup v)\ f\ x
\end{array}
$$

As long as α is finite, this definition is well-founded. As a measure function subtract the set of already visited elements (first argument of function $transclosure_{aux}$) from the (finite) set of all elements of α. It is easy to see that this measure function strictly decreases in the recursive function call. Function *transclosure* actually computes the reflexive and transitive closure of a relation as the following lemma shows.

Lemma 13 (Reflexive and transitive closure) *Assume any binary relation* $f : \alpha \to \alpha\ list$.

$$
\begin{array}{l}
\forall r.\ r \in transclosure\ f\ r\ \wedge \\
\forall r_1\ r_2\ r_3.\ r_2 \in transclosure\ f\ r_1\ \wedge\ r_3 \in transclosure\ f\ r_2 \Rightarrow r_3 \in transclosure\ f\ r_1
\end{array}
$$

As an example we define the reflexive and transitive closure of function $acl\text{-}roles^{bboard}$.

Example 12 (Reflexive and transitive closure)

$$
\begin{array}{lll}
transclosure\ acl\text{-}roles^{bboard} & Manager & = [Manager] \\
\mid & Editor & = [Editor, Manager] \\
\mid & Author & = [Author, Editor, Manager] \\
\mid & Reader & = [Reader, Author, Editor, Manager]
\end{array}
$$

Function *assignees*, which computes the assignees of a role, first computes the transitive closure of the role and then maps function members over the groups corresponding to the closure's roles.

Definition 31 (Assignees)

$$
\begin{array}{l}
assignees : group\ memory \to \alpha\ ACL \to \alpha \to subject\ list \\
assignees\ mem\ \{groups = g,\ roles = f\}\ r = \\
\quad \bigcup_{x \in transclosure\ f\ r} (\bigcup_{y \in g\ x} members\ mem\ y)
\end{array}
$$

Example 13 (Discussion bboards – Assignees)

$$
\begin{array}{lll}
assignees\ m\ ACL^{bboard''} & Manager & = [Tom, Harry] \\
\mid & Editor & = [Tom, Harry, Peter] \\
\mid & Author & = [Tom, Harry, Peter, Jenny] \\
\mid & Reader & = [Tom, Harry, Peter, Jenny]
\end{array}
$$

Function *assignees* naturally induces relation *is-subrole*, which we discuss in the following. If a role r_1 is a subrole of r_2 then r_2 dominates r_1 in the sense that if a subject is assigned role r_2 then it automatically is also assigned r_1.

Definition 32 (Is Subrole? – Non-modular)

$$\begin{aligned}&\mathit{is\text{-}subrole'} : \alpha\ \mathit{ACL} \to \mathit{group\ memory} \to \alpha \to \alpha \to \mathit{bool}\\ &\mathit{is\text{-}subrole'}\ \mathit{acl}\ \mathit{mem}\ r_1\ r_2 = \mathit{assignees}\ \mathit{acl}\ \mathit{mem}\ r_2 \subseteq \mathit{assignees}\ \mathit{acl}\ \mathit{mem}\ r_1\end{aligned}$$

Lemma 14 (*is-subrole'* is a pre-ordering) *For any ACL acl and group memory mem relation is-subrole' acl mem is a pre-ordering.*
Proof: Reflexivity and transitivity are lifted from the partial equivalence relation "subset".

At first sight this definition seems to capture our intuition properly. But at second sight you see that it heavily violates modularity. Relation *is-subrole' acl mem* crucially depends upon the surrounding group structure which basically represents the name and address book (see Section 3.2.7).

Take for example the following ACL where group *Ref 0* is assigned role x and group *Ref 1* is assigned role y. If accidentally *Ref 0* and *Ref 1* happen to consist of the same members then x is a subrole of y and vice versa. Assume now that the ACL remains totally unchanged, whereas subject "A" is added to group *Ref 0* and subject "B" is added to group *Ref 1* in the global name and address book. Suddenly neither x nor y is subrole of the other.

Changes in the name and address book, though, should not affect relation *is-subrole'*. The decision whether a role is a subrole of another ought to be taken locally in the ACL. Relation *is-subrole* remedies this weakness. As you can see from its type, it does no longer depend upon any concrete surrounding name and address book. It does not even depend upon the complete ACL of the database but rather only on the second component concerning the hierarchic structure of roles.

Definition 33 (Is Subrole?)

$$\begin{aligned}&\mathit{is\text{-}subrole} : (\alpha\ \mathit{ACL\text{-}roles}) \to \alpha \to \alpha \to \mathit{bool}\\ &\mathit{is\text{-}subrole}\ f\ r_1\ r_2 = \forall \mathit{mem}\ g.\ \mathit{assignees}\ \mathit{mem}\ (\lambda r.\{\mathit{groups} = g\ r,\ \mathit{roles} = f\ r\})\ r_2 \subseteq\\ &\qquad\qquad \mathit{assignees}\ \mathit{mem}\ (\lambda r.\{\mathit{groups} = g\ r,\ \mathit{roles} = f\ r\})\ r_1\end{aligned}$$

Lemma 15 (*is-subrole* is a pre-ordering) *Given any function f mapping roles to lists of roles, relation is-subrole f is a pre-ordering.*
Proof: Reflexivity and transitivity are lifted from the partial equivalence relation "subset".

Let's continue with our example and see which roles are subroles of other roles.

Example 14 (Discussion bboards – Is subrole?) *Figure 3.11 depicts relation is-subrole acl-roles*^{bboard}*. As you can see from the graph, this relation is – besides being a pre-ordering – also a partial equivalence relation (PER). To be precise it is even a linear ordering.* □

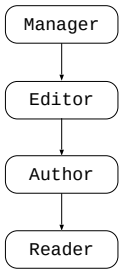


Figure 3.11: Is-subrole for bboards

It is not at all by coincidence that relation *is-subrole acl-roles*^{bboard} is a PER in the previous example. To see why, let's analyse which consequences follow from the fact that a relation *is-subrole f* is antisymmetric (by Lemma 15 it is reflexive and transitive anyway). Relation *is-subrole f* is antisymmetric if and only if

$$\forall r_1\ r_2.\ (\mathit{is\text{-}subrole}\ f\ r_1\ r_2\ \wedge\ \mathit{is\text{-}subrole}\ f\ r_2\ r_1) \Rightarrow r_1 = r_2$$

Expanding the definition of *is-subrole* and applying simple set theoretic lemmas we get the following equivalent proposition

$$\begin{aligned}\forall r_1\ r_2.\ (\forall \mathit{mem}\ g.\ \mathit{assignees}\ \mathit{mem}\ (\lambda r.\{\mathit{groups} = g\ r,\ \mathit{roles} = f\ r\})\ r_2 = \\ \mathit{assignees}\ \mathit{mem}\ (\lambda r.\{\mathit{groups} = g\ r,\ \mathit{roles} = f\ r\})\ r_1) \Rightarrow r_1 = r_2\end{aligned}$$

If a relation *is-subrole f* is not antisymmetric then there are two distinct roles that are not distinguishable (as far as their assignees are concerned) for any name and address book and any group assignment in the ACL. In other words one of these two roles is redundant. If there are no redundant roles then we call a relation minimal.

Definition 34 (*is-subrole* minimal) *Given any function f mapping roles to lists of roles. Relation is-subrole f is* ***minimal*** *if and only if it is a partial equivalence relation.*

In the above example of bboards we have seen an example of a linear relation *is-subrole f*. Defining the notion of linearity is obvious.

Definition 35 (*is-subrole* linear) *Given any function f mapping roles to lists of roles. Relation is-subrole f is* ***linear*** *if and only if it is a linear ordering.*

As you can see from above example it is not always obvious if a relation *is-subrole f* is minimal or linear, which is due to the universal quantification in the definition of *is-subrole*. It will turn out, though, that there is an alternative formulation of this relation which is decidable and hence can serve as

a decision procedure. In a sense, relation *is-subrole f* is the specification and the decision procedure is an implementation which meets the specification. In the following we present the decision procedure and show that it is both correct and complete.

Theorem 2 (Alternative definition of *is-subrole*) *Given any function f mapping roles to lists of roles. Furthermore assume that the set of roles is finite.*

$$is\text{-}subrole\ f\ r_1\ r_2 = (transclosure\ f\ r_2 \subseteq transclosure\ f\ r_1)$$

Proof: *Expanding the definition of is-subrole yields*

$$\begin{array}{l}(\forall mem\ g.\ \ assignees\ mem\ (\lambda r.\{groups = g\ r,\ roles = f\ r\})\ r_2\ \subseteq \\ \qquad\qquad assignees\ mem\ (\lambda r.\{groups = g\ r,\ roles = f\ r\})\ r_1) \ = \\ (transclosure\ f\ r_2 \subseteq transclosure\ f\ r_1)\end{array}$$

and rewriting the definition of assignees results in

$$\begin{array}{l}(\forall mem\ g.\ \bigcup_{x \in transclosure\ f\ r_2}(\bigcup_{y \in g\ x} members\ mem\ y)\ \subseteq \\ \qquad\qquad \bigcup_{x \in transclosure\ f\ r_1}(\bigcup_{y \in g\ x} members\ mem\ y)) \ = \\ (transclosure\ f\ r_2 \subseteq transclosure\ f\ r_1)\end{array}$$

Since the required equation is a boolean equation, we can split the goal into two implications.

Case 1 ("⇐") *This direction is considerably simple due to the set theoretic implication*

$$A \subseteq B \Rightarrow \bigcup_{x \in A} f\ x \subseteq \bigcup_{x \in B} f\ x$$

Case 2 ("⇒") *The converse of this set theoretic implication does not hold in this generality. It is valid, though, if restricted to injective functions f. In order to apply the converse implication, we have to find an adequate instantiation both for the name and address book mem and for the group assignment g.*

First of all we have to determine the type of subjects which we just take to be natural numbers. The name and address book is constructed as follows, where n is the cardinality of the (finite) set of roles.

$$\begin{array}{l}mem = \{objs = [(single\text{-}user\ 0\ 0),\ \ldots,\ (single\text{-}user\ (n-1)\ (n-1))], \\ \qquad refs = [0,\ \ldots,\ (n-1)]\} \\ with\ \ single\text{-}user\ s\ r = AddMember\ s\ (Admins\ [r])\end{array}$$

We take any injective function mapping roles to one-element lists of group references as group assignment function g. The existence of such an injective function is guaranteed by the finiteness of the set of roles.

Taking these particular instances of the name and address book and of the group assignment, it remains to show that the following function is injective.

$$\lambda x.\,(\bigcup_{y \in g\,x} members\ mem\ y)$$

Since g maps roles to one-element lists of group references there is trivially a function g' that maps roles to group references and which is directly connected with g by the equation $\forall x.[g'\,x] = g\,x$. Of course, injectivity is lifted directly from g to g'. Taking g' rather than g, the function to be proved injective simplifies to

$$\lambda x.\,members\ mem\ (g'\ x)$$

As injectivity is composable, it suffices to show that function members mem is injective. By construction of mem there are no circular dependencies between groups. It is easy to see, that the following lemma holds since dref mem (Ref m) = single-user m m for any $m \in \{0, \ldots, n-1\}$.

$$members\ mem\ (Ref\ m) = [m]$$

By induction on Ref we succeed in proving that members mem is injective. To be concise, we need lemma $[m] = [n] \Rightarrow m = n$, additionally. **Q.E.D.**

This theorem has simple corollaries which make it easy to decide if a relation *is-subrole f* is minimal or linear.

Corollary 1 (*is-subrole* minimal – decision procedure) *Given any function f mapping roles to lists of roles.*

$$minimal\ (is\text{-}subrole\ f) \Longleftrightarrow (\forall r_1\ r_2.\ r_1 \neq r_2 \Rightarrow transclosure\ f\ r_1 \neq transclosure\ f\ r_2)$$

Corollary 2 (*is-subrole* linear – decision procedure) *Given any function f mapping roles to lists of roles.*

$$\begin{aligned} linear\ (is\text{-}subrole\ f) \Longleftrightarrow \forall r_1\ r_2.\ & transclosure\ f\ r_1 \subseteq transclosure\ f\ r_2 \vee \\ & transclosure\ f\ r_2 \subseteq transclosure\ f\ r_1 \end{aligned}$$

With these decision procedures in hand we can return to our previous example and certify that relation *is-subrole acl-roles*^{bboard} is minimal and linear.

Example 15 (Discussion bboards – ACL minimal and linear) *In order to apply Corollaries 1 and 2, we have to recall the reflexive and transitive closure of relation acl-roles*^{bboard} *(see Example 12).*

$$\begin{array}{lll} transclosure\ acl\text{-}roles^{bboard} & Manager & = [Manager] \\ \quad | & Editor & = [Editor, Manager] \\ \quad | & Author & = [Author, Editor, Manager] \\ \quad | & Reader & = [Reader, Author, Editor, Manager] \end{array}$$

One can easily see that the conditions of minimality and linearity are satisfied by this reflexive and transitive closure.

3.2.5 States

So far we have defined the data space, the profile documents and the ACL of databases. It's time now to combine them to the state which is that part of a database that may change over time. All other parts remain invariant.

Definition 36 (States) *Assume d being the name of a database. The type of possible states for this database is $state^d$.*

$$\begin{aligned}\textbf{type}\ state^d = \{&data\text{-}space : data^d\ memory,\\ &profile : profile^d\ memory,\\ &acl\text{-}groups : role^d\ ACL\text{-}groups\}\end{aligned}$$

Please note that the acl-part of states only contains the group- but not the role-structure. As already mentioned, states represent the variable part of a database – the role structure, though, is static. We might as well have integrated the roles in the states at the price of having to use function $lift_{ACL}$ (see Definition 29) for any state update. It's a design decision to give applicability higher priority than uniformity.

Example 16 (Discussion bboards – States)

$$\begin{aligned}\textbf{type}\ state^{bboard} = \{&data\text{-}space : data^{bboard}\ memory,\\ &profile : profile^{bboard}\ memory,\\ &acl\text{-}groups : role^{bboard}\ ACL\text{-}groups\}\end{aligned}$$

When a database is created, the database contains an initial state which is uniform for all new databases. The dataspace is empty, the profile documents contain some default value and the ACL contains one single group – the database managers. Since the role representing the database managers may be disjoint in different databases, this role is – in addition to the reference to the manager group and the default profile document – a parameter for function *initial-state* which creates the initial state of a database.

Of course, we could also have omitted the profile document as parameter, starting with an empty memory as initial configuration. Again, our choice is a design decision which aims at easing the model: often it suffices to keep the initial configuration constant (see Section 5.4.3). In this case no explicit operations are required to deal with the profile documents. Easier theories and correctness proofs are the consequences.

Definition 37 (Initial States) *Assume d being the name of a database. Function $initial\text{-}state^d$ creates the initial state for some database d.*

$$\begin{aligned}&initial\text{-}state : \gamma \to \beta \to group\ Ref \to (\alpha, \beta, \gamma)state\\ &initial\text{-}state\ Manager\ p\ r = \{&data\text{-}space = new_{memory},\\ &&profile = alloc\ p\ new_{memory},\\ &&acl\text{-}groups = \lambda\, x.\ \textbf{if}\ x = Manager\ \textbf{then}\ [r]\ \textbf{else}\ []\}\end{aligned}$$

For the database of discussion bboards we have called the manager role "Manager" and in Example 11 we have determined group *Ref 0* to be the manager group. The initial set of categories is empty and eMail notification is initially disabled for all users. This leads to the following initial state for discussion bboards.

Example 17 (Discussion bboards – Initial state)

$$
\begin{aligned}
&\mathit{initial\text{-}state}^{bboard} : \mathit{state}^{bboard} \\
&\mathit{initial\text{-}state}^{bboard} = \mathit{initial\text{-}state}\ \ \mathit{Manager} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \{\mathit{categories} = [],\ \mathit{notification} = \lambda\, x.\, \mathit{false}\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Ref}\ 0)
\end{aligned}
$$

3.2.6 Operations and guards

Each database contains a set of operations in order to modify dataspace, profile documents and ACL. Some of these operations are part of the groupware system which provides a set of standard operations (as e.g. adding data or assigning groups to roles). Beyond these standard operations, each concrete application may implement additional application-specific operations (as e.g. computing the number of responses to a posting).

For our formal framework Logos we do not differentiate between standard operations of the groupware system and application-specific operations. We simply assume that for a concrete application there is a set of operations to deal with the databases.

Nevertheless, we do differentiate between two kinds of operations – depending on the impact the operation has on the database. The first kind of operations, which we call *internal*, modifies the state of the database but is invisible from outside (assuming the database to be a black box). The second kind, which we call *observable*, leaves the state of the database unchanged but renders parts of the database visible to the outer world.

Definition 38 (Internal operations) *Each database d contains a fixed number $\mathit{internal}^d$ $(0 \leq \mathit{internal}^d)$ of internal operations. Internal operations f_i^d $(1 \leq i \leq \mathit{internal}^d)$ posses the following uniform structure with function specific arity a_i $(0 \leq a_i)$.*

$$f_i^d : \mathit{subject} \to \tau_1 \to \ldots \to \tau_{a_i} \to \mathit{state}^d \to \mathit{state}^d$$

Definition 39 (Observable operations) *Each database d contains a fixed number $\mathit{observable}^d$ $(0 \leq \mathit{observable}^d)$ of observable operations. Observable operations g_i^d $(1 \leq i \leq \mathit{observable}^d)$ posses the following uniform structure with function specific arity a_i $(0 \leq a_i)$.*

$$g_i^d : \mathit{subject} \to \tau_1 \to \ldots \to \tau_{a_i} \to \mathit{state}^d \to \tau_{a_i+1}$$

We call type τ_{a_i+1} the *observable type* of the operation.

Example 18 (Discussion bboards – Operations) *In this example we continue our running example of bboards by dynamically changing the state of the discussion bboards. First we create an initial bboard and then we configure the database by constituting the set of categories in the profile. Next we edit the ACL, letting Harry assign group one editor and letting Tom assign group two author rights (see acl-groups^{bboard}'' in Example 11). Peter then writes a posting, which Jenny responses to. Finally we compute the categories of the response.*

To achieve this, we define general operations for our intended groupware system and concrete operations for our application of discussion bboards. Precisely, we define seven operations (see below). Clearly, this set of operations is not sufficient to account for bboards in their full generality, but we keep this set (artificially) small to slim our presentation. Please note that operations add-doc^{system} and assign-group-role^{system} are operations of the groupware system whereas all other operations are application specific. To ease readability we use the type abbreviation $(\alpha, \beta, \gamma)state = \{data\text{-}space : \alpha\ memory,\ profile : \beta\ memory,\ acl\text{-}groups : \gamma\ ACL\text{-}groups\}$.

$$
\begin{array}{l}
add\text{-}doc^{system} : subject \to \alpha \to (\alpha, \beta, \gamma)state \to (\alpha, \beta, \gamma)state \\
add\text{-}doc^{system}\ s\ d\ st = st\{data\text{-}space := alloc\ d\ st.data\text{-}space\} \\
\\
new\text{-}main^{bboard} : subject \to main \to state^{bboard} \to state^{bboard} \\
new\text{-}main^{bboard}\ s\ d\ = add\text{-}doc^{system}\ s\ (main\ d\{author := s\}) \\
\\
response\text{-}to^{bboard} : subject \to main\ Ref \to response \to state^{bboard} \to state^{bboard} \\
response\text{-}to^{bboard}\ s\ p\ r = add\text{-}doc^{system}\ s\ (response\ r\{parent := p\}\{author := s\}) \\
\\
map\text{-}categories^{bboard} : \\
\quad subject \to (string\ list \to string\ list) \to state^{bboard} \to state^{bboard} \\
map\text{-}categories^{bboard}\ s\ f\ st = \\
\quad st\{profile := st.profile\{categories := f\ st.profile.categories\}\} \\
\\
assign\text{-}group\text{-}role^{system} : \\
\quad subject \to group\ Ref \to \gamma \to (\alpha, \beta, \gamma)state \to (\alpha, \beta, \gamma)state \\
assign\text{-}group\text{-}role^{system}\ s\ g\ r\ st = \\
\quad st\{acl\text{-}groups := \lambda\, x.\ \textbf{\textit{if}}\ x = r\ \textbf{\textit{then}}\ (st.acl\text{-}groups\ x)\text{++}[g] \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{\textit{else}}\ st.acl\text{-}groups\ x\} \\
assign\text{-}group\text{-}role^{bboard} : \\
\quad subject \to group\ Ref \to role^{bboard} \to state^{bboard} \to state^{bboard} \\
assign\text{-}group\text{-}role^{bboard} = assign\text{-}group\text{-}role^{system} \\
\\
categories\text{-}re^{bboard} : subject \to data^{bboard}\ Ref \to state^{bboard} \to string\ list \\
categories\text{-}re^{bboard}\ s\ r\ st = (dref\ ((dref\ r\ st.data\text{-}space).parent) \\
\qquad\qquad\qquad\qquad\qquad\qquad st.data\text{-}space).categories\ \cap \\
\qquad\qquad\qquad\qquad\qquad st.profile.categories
\end{array}
$$

The operations of the groupware system are polymorphic and thus we are able

to reuse them in the context of other databases. The corresponding database specific operations add-doc^{bboard} : subject $\rightarrow$ data^{bboard} $\rightarrow$ state^{bboard} $\rightarrow$ state^{bboard} and assign-group-role^{bboard} : subject $\rightarrow$ role^{bboard} $\rightarrow$ group Ref $\rightarrow$ state^{bboard} $\rightarrow$ state^{bboard} are just instances of these polymorphic definitions.

Operation new-main^{bboard} adds a new posting and response-to^{bboard} a new response to the data space. The categories in the profile of the database are managed by operation map-categories^{bboard}. The ACL is modified by operation assign-group-role^{bboard} which assigns some group a given role. The just mentioned operations are all ***internal*** *operations, i.e. they update the state of bboards. The last operation categories-re^{bboard} is the only* ***observable*** *operation. It computes the set of categories a response is assigned to (restricted to those categories that are predefined in the profile document).*

A subtlety is involved in the definition of operation categories-re^{bboard}, concerning the type of field parent. In the declaration of type response we did not declare parent to be of type $\{\!|$ main : main, response : response $|\!\}$. Of course this would be a circular and thus invalid declaration. As long as responses only point to postings and not to responses themselves we avoid circular type definitions taking main Ref as type of parent. In order to compute the categories from a response, we first extract the pointer to the original posting and then convert the type of the reference as required (this type conversion has been omitted in the definition to ease readability).

If we were really intending to allow responses to point to the responses themselves we were stuck with our proposed solution. A solution to the problem is obvious since references are (at least in principle) nothing but natural numbers. Declaring parent to be of type nat remedies the problem of circular typing. Please note that although we have lost elegant typing for field parent, we still are able to deploy elegant typing for functions as categories-re^{bboard}.

This weakness of our model is a consequence of the weakness of the type system of HOL. Please note that there are type systems (see discussion in [34]) that do permit circular i.e. recursive type declarations. These circularities are resolved using fixed point constructions.

Let's continue with our running example. Starting form the initial state we first configure the profile of the database adding a set of categories. Next, we extend the ACL by assigning Ref 1 editor and Ref 2 author rights. To be precise, manager Harry assigns the editor and manager Tom the author rights. Finally, Peter posts the following note

$$
\begin{array}{l}
myposting : main \\
myposting = \{author = arbitrary,\ date = 03/16/2000, \\
\qquad\qquad categories = [\text{“}meetings\text{”}, \text{“}travel\text{”}], \\
\qquad\qquad note = \text{“}My\ travel\ expenses\ to\ the\ last\ meeting\ were\ 200\$\text{”}\}
\end{array}
$$

which is replied to by Jenny

$$\begin{aligned}&myresponse : main\,Ref \rightarrow response\\&myresponse\,r = \{parent = r,\ author = arbitrary,\ date = 03/17/2000,\\&\qquad\qquad note = \text{“}200\$\ is\ waste!\ I\ only\ spent\ 100\$!!\text{”}\}\end{aligned}$$

The resulting state – after application of the internal operations – is represented by the term

$$\begin{aligned}&main\text{-}response : state^{bboard}\\&main\text{-}response =\\&\quad st := initial\text{-}state^{bboard};\\&\quad st' := map\text{-}categories^{bboard}\\&\qquad\qquad Harry\,(\lambda\, xs.\ xs{+}{+}[\text{“}meetings\text{”}, \text{“}travel\text{”}, \text{“}project\text{”}])\ st\\&\quad st'' := assign\text{-}group\text{-}role^{bboard}\ Harry\,(Ref\ 1)\ Editor\ st';\\&\quad st''' := assign\text{-}group\text{-}role^{bboard}\ Tom\,(Ref\ 2)\ Author\ st'';\\&\quad st'''' := new\text{-}main^{bboard}\ Peter\ myposting\ st''';\\&\qquad response\text{-}to^{bboard}\ Jenny\,(myresponse\,(freshref\ st'''.data\text{-}space))\ st''''\end{aligned}$$

Please note that main-response.acl-groups = acl-groups^{bboard}'' (for the definition of acl-groups^{bboard}'' see Example 11). The proof of this lemma is an easy exercise.

In the end, Jenny computes the categories of her response as denoted by the term

$$categories\text{-}re^{bboard}\ Jenny\,(freshref\ st''''.data\text{-}space)\ main\text{-}response$$

which results in the list [“meetings”, “travel”]. □

So far we have not modelled access control at all. For example, Peter could have posted his note before group *Ref 1* (which is the only group he belongs to) was assigned editor rights. In the following we tackle this problem.

Any functional programming language (and the sublanguage of “executable” terms of HOL can be understood as a functional programming language) permits application of any two terms at any time – as long as parameter and argument type match. To account for access control in a programming language, any desired restriction would be programmed as conditional statement and be incorporated in the functions. In a formal logic as HOL the restriction can also be stated as a logical predicate i.e. the restriction need not necessarily be computable. Furthermore the restricting predicate, which we will call *guard* in the following, can be separated from the function itself enabling the notions of permitted and prohibited function applications. In this section we will introduce guards and in Section 3.3.3 we will see how to formalise the notions of permitted and prohibited applications of functions (or equivalently operations).

Although operations might as well have several guards, we assume for simplicity and without loss of generality[10] that there is exactly one guard per operation.

Basically a guard is just a predicate on the arguments of the operation. Any operation knows from its first argument who is liable for invoking the operation but it has no knowledge (and needs not to have any) of the surrounding context (ACL and NAB). This information, though, is required for access control. Hence the guard is stuffed with this information, too – in addition to the arguments of the operation. This additional information is represented as two functions: first a function mapping group references to sets of subjects (the members of the group) and second a function mapping roles to sets of subjects (the assignees of the role).

Definition 40 (Guards for internal operations) *Assume internal operation* $f_i^d : subject \to \tau_1 \to \ldots \to \tau_{a_i} \to state^d \to state^d$ *of database* d. *The corresponding guard* $P_{f_i}^d$ *has type*

$$P_{f_i}^d : (group\ Ref \to subject\ list) \to (role^d \to subject\ list) \to subject \to \tau_1 \to \ldots \to \tau_{a_i} \to state^d \to bool$$

Definition 41 (Guards for observable operations) *Assume observable operation* $g_i^d : subject \to \tau_1 \to \ldots \to \tau_{a_i} \to state^d \to \tau_{a_i+1}$ *of database* d. *The corresponding guard* $P_{g_i}^d$ *has type*

$$P_{g_i}^d : (group\ Ref \to subject\ list) \to (role^d \to subject\ list) \to subject \to \tau_1 \to \ldots \to \tau_{a_i} \to state^d \to bool$$

Assume any guard P_h^d (internal or observable) of type $(group\ Ref \to subject\ list) \to (role^d \to subject\ list) \to subject \to \tau_1 \to \ldots \to \tau_{a_n} \to state^d \to bool$. In case $P_h^d\ nab\ acl\ s\ x_1 \ldots x_{a_n}\ st = true$ we say that subject s is permitted to execute operation h^d with arguments $x_1, \ldots, x_{a_n}$ in state st and context *nab* and *acl*.

Example 19 (Discussion bboards – Guards) *In Example 18 we have defined the following bboard operations:* new-main^{bboard}, response-to^{bboard}, assign-group-role^{bboard}, map-categories^{bboard} *and* categories-re^{bboard}. *For all of these operations we define corresponding guards.*

- **new-main**: *This operation creates a new posting. Authors or above should be permitted to invoke the operation.*
- **response-to**: *Similar to operation* new-main^{bboard}; *responses are created, though.*

[10] If there is no guard for a function then the constant guard *false* is used. In case of multiple guards, these are transformed into a single guard by conjunction.

- **assign-group-role**: *This is the only operation for the ACL. No other subject than a manager is permitted to change the ACL, hence usage of this operation is restricted to managers only.*
- **map-categories**: *Its purpose is to predefine the set of available categories. Editors may apply this operation which has effect on the database profile.*
- **categories-re**: *So far all operations have been internal. This is the only observable operation and it computes the set of categories a response document is assigned to. We restrict access to the authors of the respective documents (please note that this decision is artificial – it was only taken to demonstrate access control for authors of documents).*

Now we define the corresponding guards[11]*.*

$$
\begin{aligned}
&P^{bboard}_{new\text{-}main} : (group\,Ref \to subject\,list) \to (role^{bboard} \to subject\,list) \to \\
&\qquad subject \to main \to state^{bboard} \to bool \\
&P^{bboard}_{new\text{-}main}\; nab\; acl\; s\; d\; st = s \in (acl\; Author) \\
&P^{bboard}_{response\text{-}to}\; nab\; acl\; s\; r\; d\; st = s \in (acl\; Author) \\
&P^{bboard}_{assign\text{-}group\text{-}role}\; nab\; acl\; s\; g\; r\; st = s \in (acl\; Manager) \\
&P^{bboard}_{map\text{-}categories}\; nab\; acl\; s\; f\; st = s \in (acl\; Editor) \\
&P^{bboard}_{categories\text{-}re}\; nab\; acl\; s\; r\; st = (s = (dref\; r\; st.data\text{-}space).author)
\end{aligned}
$$

Earlier on we asked if Peter really was permitted to post his note. Assuming the surrounding name and address book from Figure 3.6 we can now answer the question with a formal argument: Taking functions members m and assignees m $ACL^{bboard''}$ (see Example 13) to compute the surrounding context, the guard

$$
\begin{aligned}
&P^{bboard}_{new\text{-}main}\; (members\; m)\; (assignees\; m\; ACL^{bboard''})\; Peter\; myposting\; st'' = \\
&Peter \in (assignees\; m\; ACL^{bboard''}\; Author)
\end{aligned}
$$

reduces to true which means that Peter was right. At the same time, Peter was not permitted to change the ACL

$$
\begin{aligned}
&\forall g\; r.\; P^{bboard}_{assign\text{-}group\text{-}role}\; (members\; m)\; (assignees\; m\; ACL^{bboard''})\; Peter\; g\; r\; st'' = \\
&\qquad Peter \in (assignees\; m\; ACL^{bboard''}\; Manager) = false
\end{aligned}
$$

since he was no manager of the database. □

[11] To ease readability we omit some of the type declarations. The type declarations should be clear from the definitions of the operations

3.2.7 Name and address book

In groupware systems, the name and address book (NAB) is the central database where subjects and groups are stored. In order to get access to some application a user has to be registered within the NAB (except for anonymous access which has to be dealt with explicitly). For each subject a document *(personal document)* is kept which at least contains the name of the user. Often the personal document also contains key data as password or eMail address. Analogously, there is a document *(group document)* for each group listing members and subgroups. Often the group documents contain further information as e.g. the name of the administrator who created the group.

As far as our framework Logos is concerned, we store group documents but no explicit personal documents in the NAB. Personal documents are taken as a special case of group documents consisting only of a single member. In the following we nevertheless will speak of personal documents in Logos meaning the degenerated group documents.

Just as in many real groupware systems (e.g. as in Lotus Notes) the NAB is an ordinary database with personal and group documents taken as documents of the database. The dataspace is uniform for all name and address books: a collection of group documents.

Definition 42 (NAB - Data space)

$$\textbf{\textit{type}}\ \mathit{data\text{-}space}^{NAB} = \mathit{group\ memory}$$

The roles, profile documents and operations are – as for any database – freely definable within the restrictions explained in the previous sections. Each groupware system may define its own structure. The states of NABs are as expected:

Definition 43 (NAB - States)

$$\begin{aligned}\textbf{\textit{type}}\ \mathit{state}^{NAB} = \{&\mathit{data\text{-}space} : \mathit{data\text{-}space}^{NAB},\\ &\mathit{profile} : \mathit{profile}^{NAB},\\ &\mathit{acl\text{-}groups} : \mathit{role}^{NAB}\ \mathit{ACL\text{-}groups}\}\end{aligned}$$

There is one peculiarity, though, concerning initial states. In Definition 37 we have defined the dataspace of initial states to be empty. No user was then able to work with the NAB since no user was registered in the NAB. The same holds for any other database, too. The groupware system would be stuck even before it had started to become alive. For the special case of the NAB, we have to have some initial data in the dataspace – the manager of the whole application.

Definition 44 (NAB - Initial States)

$$
\begin{array}{l}
\mathit{initial\text{-}state}^{NAB} : \mathit{role}^{NAB} \to \mathit{profile}^{NAB} \to \mathit{subject} \to \mathit{state}^{NAB} \\
\mathit{initial\text{-}state}^{NAB}\ \mathit{Manager}\ p\ s = \\
\quad \{\mathit{data\text{-}space} = \mathit{single\text{-}user}_{group}\ s\ (\mathit{Ref}\ 0)\ \mathit{new}_{memory}, \\
\quad\ \mathit{profile} = p, \\
\quad\ \mathit{acl\text{-}groups} = \lambda\, x.\ \textbf{\textit{if}}\ x = \mathit{Manager}\ \textbf{\textit{then}}\ [\mathit{Ref}\ 0]\ \textbf{\textit{else}}\ []\}
\end{array}
$$

Once more we illustrate the definitions by our running example of discussion bboards. To do so, Harry creates a new groupware application in which Tom, Peter and Jenny are registered step by step. For each of them a personal document is generated in the name and address book and all of them are getting organised in a group hierarchy as indicated in Figure 3.6.

Example 20 (Discussion bboards – NAB) *The first thing to do is to generate a groupware application. As already mentioned, Harry will generate the application. To this end he will use function* $\mathit{initial\text{-}state}^{NAB}$*. All we then need to know is how the roles and the initial profile document for the NAB look like. We simply assume that the roles of the name and address book are the same as for the database of bboards (***type** $\mathit{role}^{NAB} = \mathit{role}^{bboard}$*) and that the profile is empty i.e. has type unit.*

$$\mathit{nab} = \mathit{initial\text{-}state}^{NAB}\ \mathit{Manager}\ ()\ \mathit{Harry}$$

Next, Harry registers Peter and then appoints Tom administrator for the whole application (in addition to himself). Tom in turns takes on his duty and registers Jenny.

In order to realize these steps, we need a number of operations. We reuse operation $\mathit{add\text{-}doc}^{system}$ *which we have defined polymorphically and which – as a consequence – may be instantiated for our NAB.*

$$
\begin{array}{l}
\mathit{add\text{-}doc}^{NAB} : \mathit{subject} \to \mathit{data}^{NAB} \to \mathit{state}^{NAB} \to \mathit{state}^{NAB} \\
\mathit{add\text{-}doc}^{NAB} = \mathit{add\text{-}doc}^{system}
\end{array}
$$

Beyond that we only need one further operation to add members to groups in our NAB:

$$
\begin{array}{l}
\mathit{addmember}^{NAB} : \mathit{subject} \to \mathit{subject} \to (\mathit{group}\ \mathit{Ref}) \to \mathit{state}^{NAB} \to \mathit{state}^{NAB} \\
\mathit{addmember}^{NAB}\ s\ s'\ r\ st = st\{\mathit{data\text{-}space} := \mathit{addmember}\ s'\ r\ st.\mathit{data\text{-}space}\}
\end{array}
$$

We do not spell out the guards for these operations literally, but it would be easy to fill this gap: authors of the NAB may add new groups whereas only editors are permitted to edit existing groups.

Now we are ready to define the sequence of NAB registrations[12]*:*

[12]To be precise, we also had to define personal documents for Harry, Tom, Peter and Jenny (i.e. for all single-user groups) which we have omitted for sake of simplicity

$$
\begin{array}{l}
nab' = add\text{-}doc^{NAB}\ Harry\ g_1\ nab; \\
nab'' = addmember^{NAB}\ Harry\ Tom\ (Ref\ 0)\ nab'; \\
nab''' = add\text{-}doc^{NAB}\ Tom\ g_2\ nab'' \\
\quad with \\
g_1 = Admins\ [Ref\ 0];\ AddSubgroup\ (Ref\ 0);\ AddMember\ Peter \\
g_2 = Admins\ [Ref\ 2];\ AddSubgroup\ (Ref\ 0);\ AddMember\ Jenny
\end{array}
$$

Name and address book nab''' then realizes the group memory as indicated in Figure 3.6. □

3.2.8 Object-oriented concepts

The definitions we have given so far are complete in the sense that they suffice to model groupware systems and their applications. The mechanisms we introduce in this section have no impact on the expressivity of Logos but rather improve its applicability by helping to structure operations and correctness proofs.

Object-orientation has established in software engineering as a means to structure programs and to reuse code. In Chapter "Foundations" (see 2.2.2) we have shown how to embed object-oriented structuring mechanisms as inheritance, overriding of methods, abstract methods and late-binding into HOL and how to deal with verification of such object-oriented programs. The resulting logical environment was called HOOL.

In this section we apply HOOL to Logos attaining a way to organise groupware applications along the lines of object-oriented concepts while at the same time retaining the full flexibility of verification in HOL. The definitions of Section 2.2.3 suffice to account for object-oriented structuring of groupware applications. All that remains to do is to show by means of example how to apply these mechanisms to Logos. We do not demonstrate all mechanisms but rather restrict ourselves to the most important ones: self-reference, abstract methods[13], method overriding, inheritance and late-binding.

Example 21 (Discussion bboards – Object-oriented concepts)
So far we have only dealt with discussion bboards as the only kind of databases which meant that a hierarchy of databases was non-existent. In this example we introduce some hierarchy – introducing a new database: response documents. The hierarchy is summarised in Figure 3.12.
The database of ***response documents*** *generically provides two kinds of documents: main documents and response documents. Response documents*

[13]We will use the terms "method" and "operation" as well as "class"and "database" interchangeably.

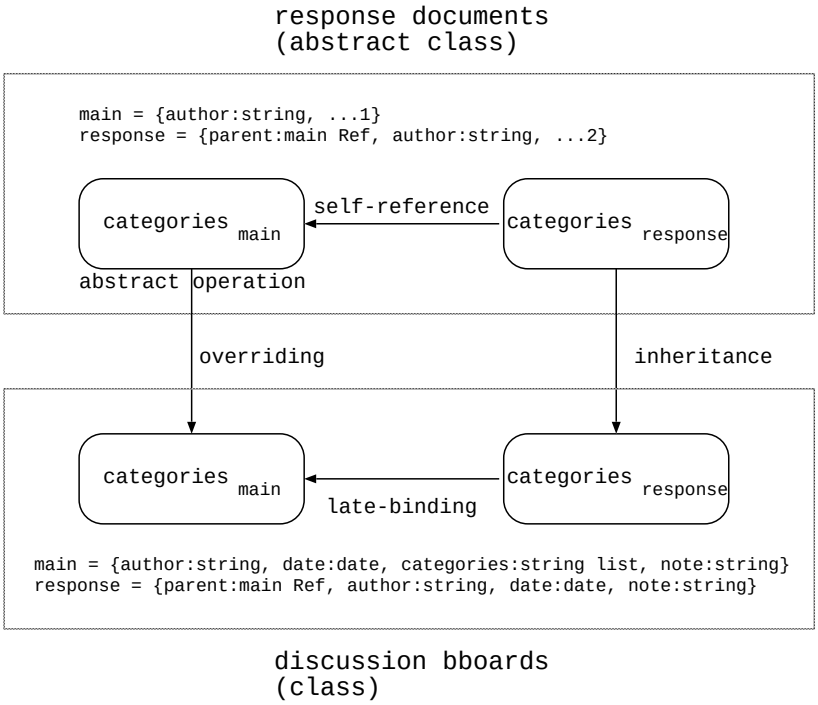


Figure 3.12: Object-oriented concepts

can only coexist with their corresponding main documents. In this sense, each response document contains a link to its main document. The first thing we do is to define the data, data space and state of this database.

There is not much structure inherent in the data of response documents. All we postulate is that the name of the author is stored both for main documents and response documents and that response documents contain a reference pointing to the corresponding main document.

$$\begin{array}{ll} \textbf{\textit{record}}\ main = & \textbf{\textit{record}}\ response = \\ \quad author : string & \quad parent : main\ Ref \\ & \quad author : string \end{array}$$

In Section 2.2.1 we have explained how records are translated to HOL (using a shallow encoding). Recall that main and response are declared as extensible records, i.e. we may use types $main = \{author : string, \ldots\}$ *and* $response = \{parent : main\ Ref, author : string, \ldots\}$ *from now onwards.*

As we have said, the database of response documents contains two kinds of documents: main documents and response documents. This is reflected in the definition of the data by a co-record composed of main and response.

$$(\ldots_1, \ldots_2)\, data^{response} =$$
$$\quad \{|\ main : \{author : string, \ldots_1\},$$
$$\quad\ \ response : \{parent : main\, Ref,\ author : string, \ldots_2\}\ |\}$$

Admittedly, the notation $\ldots_1$ and $\ldots_2$ is quite unusual, but this is a simple consequence from our decision in Section 2.2.1 to introduce "..." as a syntactic part of HOOL (in particular as abbreviation for type variables). The extensible records main and response might be instantiated differently in "subclasses" – hence we need subscripts to differentiate them.

Once you got accustomed to the unusual notation, you will find the type of states quite standard. Since we do not want to anticipate the profile documents and the roles at all, the states are polymorphic in their types.

$$\begin{aligned}\textbf{type}\ (\ldots_1, \ldots_2, \alpha, \beta)\, state^{response} = \\ \{&data\text{-}space : (\ldots_1, \ldots_2)\, data^{response}\ memory,\\ &profile : \alpha,\\ &acl\text{-}groups : \beta\ ACL\text{-}groups\}\end{aligned}$$

Next we introduce our first operation: $categories^{response}_{main}$. Its purpose is to compute the categories a main document is assigned to. In contrast with the next operations it is abstract, i.e. no implementation is given. Please note that implementations would not make sense at this stage either since main documents at this level of the hierarchy do not make any statement about their categories.

$$\begin{aligned}categories^{response}_{main} : subject \to (\ldots_1, \ldots_2)\, data^{response}\ Ref \to \\ (\ldots_1, \ldots_2, \alpha, \beta)\, state^{response} \to string\ list\end{aligned}$$

Taking any implementation of $categories^{response}_{main}$ for granted – referring to methods (operations) of the same class (database) is often called "self-reference" in object-oriented programming languages – we can define an operation $categories\text{-}re^{response}$, which computes the set of categories a response document is assigned to. Basically, it first picks out the corresponding main document and then applies operation $categories^{response}_{main}$ to this document.

$$\begin{aligned}&categories\text{-}re^{response} : subject \to (\ldots_1, \ldots_2)\, data^{response}\ Ref \to \\ &\qquad\qquad (\ldots_1, \ldots_2, \alpha, \beta)\, state^{response} \to string\ list\\ &categories\text{-}re^{response}\ s\ r\ st = categories^{response}_{main}\ s\ (dref\ r\ st.data\text{-}space).parent\ st\end{aligned}$$

As far as our example is concerned, we have concluded the database of response documents. Let's progress now in our hierarchy and redefine discussion bboards on the basis of response documents. In the database of bboards we want to override the abstract operation $categories^{response}_{main}$ with a concrete implementation for bboards. Operation $categories\text{-}re^{response}$ gets inherited from the response documents falling back upon the newly implemented $categories^{response}_{main}$ due to late-binding.

Overriding the abstract method $categories_{main}^{response}$ *amounts to defining it for concrete instantiations of the type variables. Overloading permits different implementations in different levels of the hierarchy provided the type-instantiations differ. We instantiate the type-variables to achieve the same types of data and data space as in Example 8.*

$$\begin{array}{l}
\textbf{type}\ ext_1 = (\{date : date,\ categories : string\ list,\ note : string\},\\
\qquad\qquad \{date : date,\ note : string\})\\
\textbf{type}\ ext_2 = (ext_1,\ profile^{bboard},\ role^{bboard})\\
categories_{main}^{response}\ s\ (r : (ext_1\ data^{response})\ Ref)\ (st : ext_2\ state^{response}) =\\
\qquad (dref\ r\ st).categories \cap st.profile.categories
\end{array}$$

The following trivial lemma shows that late-binding really works in Logos:

Lemma 16 (Late-binding)

$$\begin{array}{l}
categories\text{-}re^{bboard}\ s\ r\ st \subseteq st.profile.categories\\
\quad with\\
categories\text{-}re^{bboard}\ s\ r\ st =\\
\quad categories\text{-}re^{response}\ s\ (r : (ext_1\ data^{response})\ Ref)\ (st : ext_2\ state^{response})
\end{array}$$

Please note that instantiation ext_2 *is more specific than needed – for the profile documents and the roles we have unnecessarily chosen particular, concrete types. All that is required is that the profile documents contain a document categories* : *string list. A more general instantiation would use the following type instantiation* ext_3 *in spite of* ext_2*.*

$$\textbf{type}\ ext_2 = (ext_1,\ \{categories : string\ list\ \ldots\},\ \beta)$$

Using overloading to express late-binding is pretty unusual and has some quite unexpected consequences. Looking at operation $categories\text{-}re^{response}$ *you will notice that it is polymorphic in operation* $categories_{main}^{response}$*. The realization of this feature, though, is quite different from what you probably would expect: polymorphism by higher-order functions (see e.g. [12] where late-binding really has been modelled by higher-order functions). Overloading paves the way to write higher-order functions without having higher-order functions explicitly at hand. We believe that this is one of the reasons why late-binding has become so popular in object-oriented programming languages: it allows (at least to some extend) for the expressiveness of higher-order functions without having to enrich the language with explicit higher-order functions. Furthermore it enables users to use higher-order functions without ever having heard of this powerful mechanism.* □

Please note that guards (see Definitions 40 and 41) are ordinary functions in HOL and hence are subject to object-oriented concepts just as operations.

Excursion: Dependent types

In the previous definitions we have freely indexed over the name of the database. For example, we have stated in Definition 36:

$$\begin{aligned}\textbf{type}\ \mathit{state}^{d} = \{&\mathit{data\text{-}space} : \mathit{data}^{d}\ \mathit{memory},\\ &\mathit{profile} : \mathit{profile}^{d}\ \mathit{memory},\\ &\mathit{acl\text{-}groups} : \mathit{role}^{d}\ \mathit{ACL\text{-}groups}\}\end{aligned}$$

To be honest, this is not a valid definition in HOL since in HOL there is no mechanism to instantiate "d" with the name of the desired database. The reason is that HOL possesses a simple type system which does not permit any dependencies between types and terms. This is no real obstacle, though, since our definition can be read as a meta-definition (or more precisely meta definition-scheme) with valid HOL-instantiations in applications. Being an instance of above meta-definition, the definition

$$\begin{aligned}\textbf{type}\ \mathit{state}^{\mathit{bboard}} = \{&\mathit{data\text{-}space} : \mathit{data}^{\mathit{bboard}}\ \mathit{memory},\\ &\mathit{profile} : \mathit{profile}^{\mathit{bboard}}\ \mathit{memory},\\ &\mathit{acl\text{-}groups} : \mathit{role}^{\mathit{bboard}}\ \mathit{ACL\text{-}groups}\}\end{aligned}$$

from Example 16 is a valid definition in HOL.

Had we taken a type theory with dependent types (as e.g. Calculus of Constructions – see [20]) as formal basis for Logos, we would have been able to formalise the meta-definitions in the logic itself. The definition of state^{d} would then have been rewritten as

$$\begin{aligned}&\mathit{state} : \mathit{database\text{-}names} \to *\\ &\mathit{state}\ d = \{\mathit{data\text{-}space} : (\mathit{data}\ d)\ \mathit{memory},\\ &\qquad\qquad \mathit{profile} : (\mathit{profile}\ d)\ \mathit{memory},\\ &\qquad\qquad \mathit{acl\text{-}groups} : (\mathit{role}\ d)\ \mathit{ACL\text{-}groups}\}\end{aligned}$$

with *database-names* being some universe for database names. Please note that no corresponding type exists in Logos since the set of database names is part of the meta-language and not part of Logos itself.

The declaration $\mathit{state} : \mathit{database\text{-}names} \to *$ expresses that *state* is a type constructor which, given a database name, returns a type (by the way, *data* and *role* are type constructors of the same kind). In the definition of *state* the defining term is abstracted over the database name.

As a consequence, the following definition of $\mathit{state}^{\mathit{bboard}}$ is a valid type definition in a type theory with dependent types – applying type constructor *state* to *bboard* (assuming $\mathit{bboard} : \mathit{database\text{-}names}$):

$$\begin{aligned}&\mathit{state}_{\mathit{bboard}} : *\\ &\mathit{state}_{\mathit{bboard}} = \mathit{state}\ \mathit{bboard}\end{aligned}$$

Although expression "*bboard*" occurs twice in this definition, its meaning is completely different in each case. On the right hand side of the equation "*bboard*" occurs as a typed term of the logic, whereas the same expression on the left hand side of the equation is a syntactic sub-term with "*bboard*" typeset in subscript (rather than typesetting "*statebboard*") only to ease reading in the presentation.

Being able to represent database names within Logos would also have consequences for the operations. Function *initial-state*, for example, would be assigned the following type.

$$\mathit{initial\text{-}state} : \forall d.\ \mathit{role}\ d \to \mathit{profile}\ d \to \mathit{group}\ \mathit{Ref} \to \mathit{state}\ d$$

The implementation of *initial-state* would have to be abstracted over the name of the database – in addition to the abstractions of the former Definition 37. Defining a function to generate the initial state of discussion bboards would amount to functional application: *initial-state bboard*.

In a sense, these remarks justify our previous meta-definitions with true HOL-instantiations in Logos. In contrast with dependent types, this instantiation can only take place on a meta-level, whereas it could take place within the logic, if we had chosen dependent types as foundation for Logos.

The superiority of type theories with dependent types over the type system of HOL in the just mentioned sense might suggest to use such a type theory rather than HOL as formal basis for Logos. As far as the expressivity of definitions is concerned, the argument is justified. In particular we would come up against limiting factors if we tried to prove meta-theory of Logos within HOL itself. All meta-definitions would evade such investigations in HOL – even though such investigations still were possible on a meta-mathematical level, i.e. beyond HOL.

Experience (see [12, 22, 23, 24, 25, 26]) has shown that the expressiveness of systems with dependent types in turn is a burden in concrete applications. When dealing with applications pragmatically, systems that implement the simple type system of HOL (like Isabelle/HOL [28]) are superior to systems that implement dependent types theories (like LEGO [36] or Coq [5]).

3.3 Groupware systems and applications

So far we have dealt with single databases but we have not made explicitly clear how to deal with particular groupware systems or concrete applications consisting of a number of databases. At some points we have pointed out already that from the Logos point of view groupware systems and applications do not make much of a difference: both of them provide databases including data spaces and operations.

From a more practical point of view, though, groupware systems and applications differ. A groupware system is implemented only once and then applied for different purposes. Its correctness is proven (or at least investigated) once and forever. The situation is completely different for concrete applications. For each application a new implementation including correctness proof is required.

Later in this paper we will deal with both issues (groupware systems and applications) explicitly. In Chapter 4 we show how to model the key authorisation concept of Lotus Notes in Logos. In the subsequent chapter we refine this model and present a concrete application of discussion bboards shared by several teams.

For this section we do not differentiate between groupware systems and applications. An application is viewed as a concrete extension of a groupware system by application specific peculiarities. In case of this extension being empty, the just given notion of "application" collapses with the notion of "groupware system". We therefore use the terminology "application" uniformly to denote true applications as well as groupware systems.

Any application consists of a number of databases. The state of a concrete application is the product of the states of its databases. Transitions of the application's state are regulated by access control mechanisms.

3.3.1 Databases

Although we have defined all constituents of databases already in the last section, we will have a closer look at databases again. As a repetition recall that any database comprises:

- **Dataspace:** Collection of data
- **Profile Documents:** Configuration
- **ACL:** Access rights
- **Operations:** Dynamic behaviour
- **Guards:** Access control

For each of these notions we have given a complete definition in HOL, but we have not given a precise HOL definition for databases as a whole yet. The reason is that there are a number of alternatives – each of them with far-reaching consequences. We discuss two extreme alternatives and vote for one of them which we will follow throughout this paper.

The major question that arises is which components may be subject to change and which are invariant. An extreme position was to have no invariant components at all. In this case even the implementation (i.e. the operations) could be changed by users. Clearly, this model would be most

flexible at the prize of complicating all definitions heavily. Furthermore, the type system of HOL would not be sufficient. A type system with dependent types (as e.g. [20]) would be required to achieve full flexibility.

From a more pragmatic point of view most applications can cope without this full generality. For these applications a simpler solution suffices and – which is much more important – is much more manageable. With our decision to choose HOL rather than a sophisticated type theory with dependent types we have already preferred pragmatic arguments to theoretical ones. It is only consequent to have as many static components as possible and to reduce the variable part of a database to a minimum. In this sense we have defined the state (i.e. the variable part) only to consist of the dataspace, the profile documents and the group assignment in the ACL. We believe that this is the minimal configuration needed for true applications while at the same time covering most cases. We will back up this claim in Chapter 5 where we develop a non-trivial application of shared bboards in Logos.

To summarise, there is no single HOL definition for databases. Each database comprises a set of HOL definitions – each of them describing a particular aspect of the database. Combining these aspects into a single definition is only possible on a meta-level.

Please note, that Section 3.2.8 substantially benefits from this decision. Modelling late-binding by overloading requires individual HOL definitions for each operation and would fail if all operations were packed together in a vector.

3.3.2 States

Just as we have reduced the variable part in single databases to a minimum, we do the same for whole applications. Although it would be in principle possible to change the mixture of databases while execution (as it is feasible in real groupware systems like Lotus Notes) we assume that the set of databases is invariant (for an example of a dynamically changing database structure see the case study in Chapter 5 where we allow dynamic generation of a particular kind of databases – namely the so-called local NABs).

The states of applications hence are records with the states of the single databases as fields. Each application contains at least one particular database: the name and address book.

Definition 45 (States) *Assume $d_1, \ldots, d_n$ being the names of databases which application a consists of. The type of possible states for this application is $state^a$.*

$$\textbf{\textit{type}}\ state^a = \{nab : state^{NAB}, st^{d_1} : state^{d_1}, \ldots, st^{d_n} : state^{d_n}\}$$

Allowing the mixture of databases to change while execution still would require that the set of possible databases was fixed i.e. no new kinds of

databases were implemented while execution – only new instances of already existing databases were derived. The set of possible databases (to be precise their states) would have to be summarised into one single sum type *all-states*.

$$\textbf{type}\ state^a = \{nab : state^{NAB},\ st : all\text{-}states\ list\}$$

The requirement of a fixed set of possible databases even could be dropped using existential types (see [17, 18] for first-class abstract types) assigning *st* type $(\exists\alpha.\alpha)\ list$. But unfortunately, existential types are not incorporated into HOL.

In Section 3.2.6 we have defined operations for single databases. Lifting these operations to whole applications is generic and simple.

Definition 46 (Lifting internal operations) *Assume d being the name of a database and* $f_i^d : subject \to \tau_1 \to \ldots \to \tau_n \to state^d \to state^d$ *being an internal operation. The corresponding lifted operation* $f_i^{a,\,d}$ *(lifted to application a) is defined as follows:*

$$\begin{array}{l} f_i^{a,\,d} : subject \to \tau_1 \to \ldots \to \tau_n \to state^a \to state^a \\ f_i^{a,\,d}\ s\ x_1\ \ldots\ x_n\ st = lift^{a,\,d}\ (f_i^d\ s\ x_1\ \ldots\ x_n) \\ \quad with \\ lift^{a,\,d} : (state^d \to state^d) \to (state^a \to state^a) \\ lift^{a,\,d}\ f\ st = st\{st^d := f\ st.st^d\} \end{array}$$

Definition 47 (Lifting observable operations) *This definition is analogous to the lifting of internal operations, though using the following lifting function instead.*

$$\begin{array}{l} lift^{a,\,d} : (state^d \to \tau_{n+1}) \to (state^a \to \tau_{n+1}) \\ lift^{a,\,d}\ f\ st = f\ st.st^d \end{array}$$

In the previous sections we have developed some pieces of a jigsaw but we have not shown how the pieces fit together. In the following example we will put the pieces together – a uniform picture appears. To keep indices small, we call the application of bboards simply "*b*".

Example 22 (States of applications) *In this example we first perform the registrations from Example 20 and then deal with the database of bboards as shown in Example 18.*

$$
\begin{aligned}
state_1 &:= \{nab = initial\text{-}state^{NAB}\ Manager\ ()\ Harry,\\
&\qquad bboard = initial\text{-}state^{bboard}\};\\
state_2 &:= (add\text{-}doc^{b,\,NAB}\ Harry\ g_1)\ state_1;\\
state_3 &:= (addmember^{b,\,NAB}\ Harry\ Tom\ (Ref\ 0))\ state_2;\\
state_4 &:= (add\text{-}doc^{b,\,NAB}\ Tom\ g_2)\ state_3;\\
state_5 &:= (map\text{-}categories^{b,\,bboard}\ Harry\ (\lambda\, x.\ x{+}{+}C))\ state_4;\\
state_6 &:= (assign\text{-}group\text{-}role^{b,\,bboard}\ Harry\ (Ref\ 1)\ Editor)\ state_5;\\
state_7 &:= (assign\text{-}group\text{-}role^{b,\,bboard}\ Tom\ (Ref\ 2)\ Author)\ state_6;\\
state_8 &:= (new\text{-}main^{b,\,bboard}\ Peter\ myposting)\ state_7;\\
state_9 &:= (response\text{-}to^{b,\,bboard}\ Jenny\ (myresponse\ (Ref\ 0)))\ state_8;
\end{aligned}
$$

with

$$
\begin{aligned}
g_1 &= Admins\ [Ref\ 0];\ AddSubgroup\ (Ref\ 0);\ AddMember\ Peter\\
g_2 &= Admins\ [Ref\ 2];\ AddSubgroup\ (Ref\ 0);\ AddMember\ Jenny\\
C &= [\text{“}meetings\text{”}, \text{“}travel\text{”}, \text{“}project\text{”}]
\end{aligned}
$$

The resulting state of the application is depicted in Figure 3.13. □

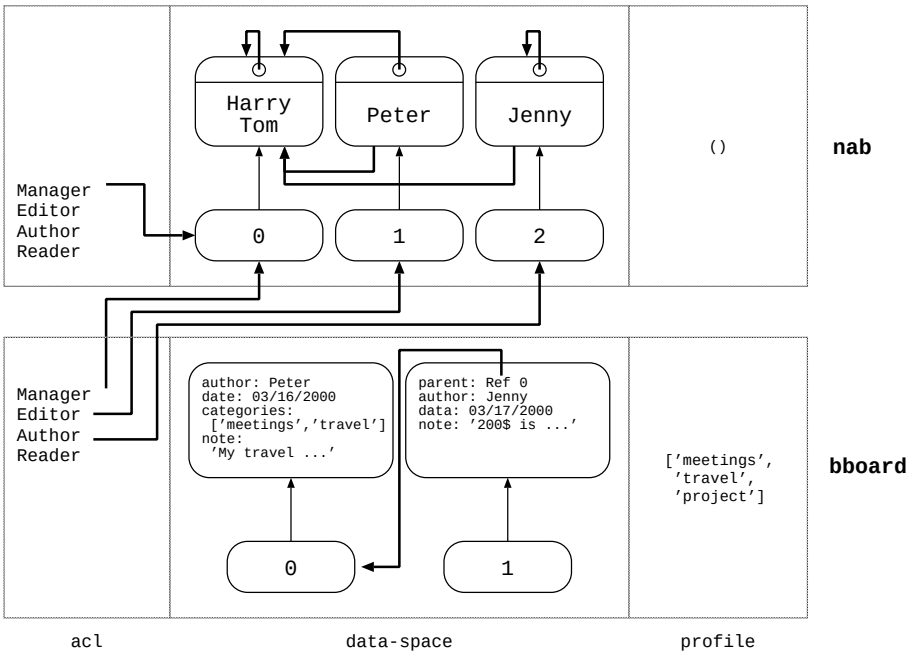


Figure 3.13: State of application

3.3.3 Access control

In Section 3.2.6 we have done solid groundwork for a comprehensive model of access control. For each operation we have defined a corresponding guard

that decides under which circumstances the operation may be used by some subject. In the definition of guards we have taken functions (the context) of type *group Ref* $\rightarrow$ *subject list* (NAB) and $role^d \rightarrow$ *subject list* (ACL) as input without explaining in detail how to generate input functions of this kind. When lifting guards to applications (see subsequent definition) we fill this gap.

Definition 48 (Lifting guards) *Assume any guard* $P_h^{d_k}$ *for internal or observable operation* h^{d_k} *of database* d_k *with input types (group Ref* $\rightarrow$ *subject list),* $(role^{d_k} \rightarrow$ *subject list), subject,* $\tau_1, \ldots, \tau_n$ *and* $state^{d_k}$*. The lifted guard (lifted to application a) is called* $P_h^{a,\,d_k}$ *and has type*

$$P_h^{a,\,d_k} : subject \rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow state^a \rightarrow bool$$

It is defined as

$$\begin{aligned} P_h^{a,\,d_k}\ s\ x_1\ \ldots\ x_n\ st = P_h^{d_k}\ & (members\ st.nab.data\text{-}space) \\ & (assignees\ st.nab.data\text{-}space\ acl) \\ & s\ x_1\ \ldots\ x_n\ st.st^{d_k} \end{aligned}$$

with

$$acl = \{groups = st.st^{d_k}.acl\text{-}groups,\ roles = acl\text{-}roles^{d_k}\}$$

Please note that $P_h^{a,\,d_k}$ does – in contrast with $P_h^{d_k}$ – not depend upon any context. The context is hidden in the NAB of the application and in the ACL of the database, equally.

Example 23 (Lifting guards) *In Example 19 we have defined guard* $P^{bboard}_{new\text{-}main}$ *for operation* $new\text{-}main^{bboard}$*. We have mentioned that this guard depends upon the surrounding context. The surrounding name and address book and access control list, though, was only sketched. Using lifted guards we confirm that Peter was permitted to post his note:*

$$\begin{aligned} & P^{b,\,bboard}_{new\text{-}main}\ Peter\ myposting\ state_7 = \\ & P^{bboard}_{new\text{-}main}\ \ (members\ state_7.nab.data\text{-}space) \\ & \qquad\qquad (assignees\ state_7.nab.data\text{-}space\ acl) \\ & \qquad\qquad Peter\ myposting\ state_7.bboard = \\ & Peter \in (assignees\ state_7.nab.data\text{-}space\ acl\ Author) = true \\ & \quad with \\ & acl = \{groups = state_7.bboard.acl\text{-}groups,\ roles = acl\text{-}roles^{bboard}\} \end{aligned}$$

□

Lifting guards to applications has lead us one step further but we have not hit the target yet. We are able now to express and prove that single steps of the application meet some requirements. In order to prove properties of the whole application, we first have to determine the potential steps of the application. To this end a "closed-world assumption" is used: the set

of operations and guards of the application is enumerated – the closed-world assumption guarantees that the application exactly consists of these operations and guards. Formally, we define an inductive set: the set of potential steps (or transitions, if you like).

Definition 49 (Transition relation) *Assume any application a. The inductive set* $transition^a : (state^a \times subject \times state^a)set$ *is defined by the following introduction rules (we write* $st \xrightarrow{s}_a st'$ *rather than* $(st, s, st') \in transition^a$*).*

For each database d_k *and for each operation* $f^{a,\,d_k}$ *of the database (with corresponding guard* $P_f^{a,\,d_k}$*) define an introduction rule.*

$$\frac{P_f^{a,\,d_k}\ s\ x_1\ \ldots\ x_n\ st}{st \xrightarrow{s}_a f^{a,\,d_k}\ s\ x_1\ \ldots\ x_n\ st}$$

Additionally define a rule $st \xrightarrow{s}_a st$ *(reflexivity of the transition relation) which expresses that any subject s may invoke no operation i.e. leave the state st unchanged at any time.*

We read $st \xrightarrow{s}_a st'$ as: subject s may transform state st of the application a to state st'.

Example 24 (Transition relation) *Among the many introduction rules for bboards we pick the one for operation* $new\text{-}main^{bboard}$ *of database bboard. See Example 19 for the definition of* $P^{bboard}_{new\text{-}main}$*.*

$$\frac{P^{b,\,bboard}_{new\text{-}main}\ s\ x\ st}{st \xrightarrow{s}_b new\text{-}main^{b,\,bboard}\ s\ x\ st}$$

All variables in this introduction rule for $new\text{-}main^{bboard}$ *are universally quantified. Instantiating subject s with Peter, posting x with myposting and the state st with* $state_7$ *we get a precondition, which we already encountered:* $P^{b,\,bboard}_{new\text{-}main}$ *Peter myposting* $state_7$*. In Example 23 we have answered this precondition positively already.*

Example 24 demonstrates that we are able now to express and verify the correctness of single transitions. Furthermore we know from the inductive definition of $transition^a$ which transitions the application consists of. All that remains is to put the pieces together and to define the set of *admissible states* of the application (i.e. the states that can be reached from admissible initial states applying admissible transitions only).

Definition 50 (Admissible states) *Assume some application a. The set of admissible states* $admissible^a : state^a$ *set for this application is defined by the following introduction rules.*

All initial states of the application are admissible. Initial states of an application consist of all initial states admissible for single databases.

$$\begin{aligned}\{&nab = \mathit{initial\text{-}state}^{NAB}\ \mathit{Manager}_{NAB}\ p_{NAB}\ s_{NAB},\\ &st^{d_1} = \mathit{initial\text{-}state}\ \mathit{Manager}_{d_1}\ p_{d_1}\ r_{d_1},\\ &\ldots\\ &st^{d_n} = \mathit{initial\text{-}state}\ \mathit{Manager}_{d_n}\ p_{d_n}\ r_{d_n}\} \in \mathit{admissible}^a\end{aligned}$$

The second introduction rule expresses that admissible states are invariant under transitions of relation transition^a.

$$\frac{st \in \mathit{admissible}^a \quad \wedge \quad st \xrightarrow{s}_a st'}{st' \in \mathit{admissible}^a}$$

Example 25 (Admissible states) *Defining the set of admissible states for bboards is straightforward. See Example 22 for the definition of state$_1$.*

$$\mathit{state}_1 \in \mathit{admissible}^b \qquad \frac{st \in \mathit{admissible}^b \quad \wedge \quad st \xrightarrow{s}_b st'}{st' \in \mathit{admissible}^b}$$

In a remark to Definition 49 we have stated that we read $st \xrightarrow{s}_a st'$ as: subject s may transform state st of the application a to state st'. Assume st not being admissible. In this case we wouldn't call such a transition admissible. The definition of admissible transitions takes care of this issue.

Definition 51 (Admissible transitions) *Assume some application a. Relation $\Longrightarrow_a$ of admissible transitions is defined as*

$$st \overset{s}{\Longrightarrow}_a st' = st \in \mathit{admissible}^a \wedge st \xrightarrow{s}_a st'$$

Analogously we define a number of transition relations for admissible observations.

Definition 52 (Admissible observations) *Assume some application with name a. For each observable type τ (i.e. each type which is observable type of some observable operation) define an inductive transition relation $\Longrightarrow_{a,\tau}$: $(\mathit{state}^a \times \mathit{subject} \times \tau)\mathit{set}$. For each observable operation g^{d_k} of database d_k with observable type τ state an introduction rule:*

$$\frac{st \in \mathit{admissible}^a \quad \wedge \quad P_g^{a,\,d_k}\ s\ x_1\ \ldots\ x_n\ st}{st \overset{s}{\Longrightarrow}_{a,\tau} g^{a,\,d_k}\ s\ x_1\ \ldots\ x_n\ st}$$

Example 26 (Admissible transitions and observations) *All of the transitions (state_i to state_{i+1}) from Example 22 are admissible. Applying function categories-re$^{b,\,bboard}$ Jenny (Ref 1) to state$_9$ also is admissible i.e. Jenny may compute the categories of her response provided the application is in state$_9$.* □

All definitions we have introduced so far serve the same purpose: modelling access control for groupware applications. What we have not shown yet is how to deploy these definitions to state properties of applications and how to prove their correctness w.r.t the properties. The following simple lemma fills this gap.

Example 27 (Discussion bboards – Authors registered in NAB) *First we state the proposition to be proved. It expresses that all subjects entered in the author fields of postings or responses are registered in the NAB.*

$$\begin{array}{l} \forall st \in admissible^b.\ \forall d \in objects\,(st.bboard.data\text{-}space). \\ \quad d.author \in \bigcup_{g \in objects\,groups} members_{aux}\,(Some\,g)\,groups \\ with \\ \quad groups = (st.nab.data\text{-}space) \end{array}$$

The proof first requires induction on the set of admissible states and then elimination of the relation transition^b.

The base case for the induction on the set admissible^b is conceivably simple: the dataspace of state$_1$ is empty. For the induction step we may assume that $st \in admissible^b$ and that $st \xrightarrow{s}_b st'$. As we have some transition in our assumptions we may use elimination which basically amounts to case distinction. In each case we may assume that the corresponding guard holds.

Case 1: $st' = new\text{-}main^{b,\,bboard}\ s\ d'\ st$
As we have said, we may assume that the corresponding guard holds i.e.

$$\begin{array}{l} P^{bboard}_{new\text{-}main}\,(members\ st.nab.data\text{-}space) \\ \qquad\qquad (assignees\ st.nab.data\text{-}space\ acl) \\ \qquad\qquad s\ d'\ st.bboard = s \in (assignees\ st.nab.data\text{-}space\ acl\ Author) \\ with\ acl = \{groups = st.bboard.acl\text{-}groups,\ roles = acl\text{-}roles^{bboard}\} \end{array}$$

This case follows from

$$\begin{array}{l} assignees\ st.nab.data\text{-}space\ acl \subseteq \\ \quad \bigcup_{g \in objects\,(st.nab.\,data\text{-}space)} members_{aux}\,(Some\,g)\,(st.nab.data\text{-}space) \end{array}$$

together with

$$\begin{array}{l} objects(\,st'.bboard.data\text{-}space) = \\ \quad d'\{author := s\} \bigcup objects(\,st.bboard.data\text{-}space) \end{array}$$

Case 2: $st' = response\text{-}to^{b,\,bboard}\ s\ r\ d\ st$
Similar to Case 1.

Cases 3–5: *Trivial since the operations do not affect the dataspace of bboard.*

□

Chapter 4

Modelling Lotus Notes in Logos

As far as groupware solutions are concerned, there are essentially three competitors which share the market: Lotus Notes, Microsoft Exchange and Novell Groupwise – in the given order. In Europe, for example, over 50% of all groupware systems that have been installed in the first quarter of the year 2000, have been based on Lotus Notes; another 38% have been based on Microsoft Exchange.

All three of them have been used for a long time, but nevertheless they are quite different with respect to their functionality. Over the years, Lotus Notes has evolved from a simple database application to a comprehensive groupware application platform which offers much more than ordinary workgroup functionality (like email, calender or task and resource management) and which can be used as basis for knowledge management in large enterprises. In contrast with Lotus Notes, Microsoft Exchange and Novell Groupwise are developed by companies which are mainly focused on operating systems. Likewise restricted are their functionalities.

In the last chapter we have proposed Logos as a model for authorisation in groupware systems. Clearly, a theoretical model cannot capture all subtleties involved in such a wired system as Lotus Notes or Microsoft Exchange. Trying to find a complete model for these real life systems is hopeless from the beginning. Nevertheless, real-life security-critical applications provoke the need for at least some formal methods to improve their quality (being aware that full verification is far beyond what can be achieved). Even a formal language in which to specify central aspects of security critical applications would be a huge step towards a more systematic and hence less error-prone software-engineering. We do not claim that Logos is a ready to use product to this end, but we would like to push forward towards a more formal treatment of access control in real-life

groupware applications.

In this section we focus on Lotus Notes – the leading system in the groupware market. We have not chosen upon this system for its dominance in the market but for its flexibility with respect to security critical applications. Being a flexible platform in which to implement applications with sophisticated levels of access rights distinguishes Lotus Notes from the competitors and makes it an ideal candidate for our investigations.

4.1 About Lotus Notes

Lotus Notes may well be called "pioneer" with respect to groupware systems. It was the first system to support computer based cooperation between working groups. Nowadays, Lotus Notes is a modern messaging system, which aims at efficient communication, information exchange and knowledge management and thus is used to support project work and intranet or extranet solutions.

4.1.1 Functionality

Although Lotus Notes provides a huge variety of functionalities, the key functionalities can be categorised as follows.

- **Data storage**: Documents in a Lotus Notes database may contain objects that belong to quite a number of datatypes – including text, rich text, numbers, pictures, audio and video files and many more. The integrated full-text search engine permits indexing and searching of documents. The presentation of the stored objects can be determined dynamically, depending on the user's name, status or personal preferences.

- **Directory**: All information concerning servers, configuration, administration, user registration and security are stored in a single database (directory) which can be dealt with like any other database. The directory is the basis for robust and secure applications.

- **Replication**: The purpose of replication is to synchronise information (documents) and application (databases) that are spread spatially. Replication concerns all aspects of applications down to the smallest entities like fields.

- **Messaging**: Messaging comprises working group functionalities such as shared project management, calendars, newsgroups, discussion bboards and email, which is the core functionality for intranets in enterprises. All these messaging tools are implemented as ordinary

databases in the framework Lotus Notes and hence may be adapted freely for concrete applications.

- **Integration**: Lotus Notes, which itself is not a relational database, may be connected to relational databases, transaction systems or application programs. To this end it supports a wide variety of standards as e.g. CORBA.
- **Workflow**: A special workflow engine helps to distribute documents, to forward them and to track their way as they process in a workflow. Part of the workflow support is also a version control.
- **Agents**: Agents are program parts the help to automatise repeatedly occurring tasks. They may be triggered manually, by events or periodically. Recently, a java virtual machine has been integrated to support java agents.

This functionality is spread over a number of tools which can be used to implement and administrate applications that are tailored to the demands of the customers.

4.1.2 Components

Lotus Notes is a client-server architecture – hence the server plays a vital role for a robust application. The *Lotus Notes server* is administrated by means of commands edited in a console (local or remote). Typical tasks that are performed by an administrator are broadcast messages, releasing resources or shutting down the server. Since administrating a server on a console is cumbersome, there is a special client called *administrator client* which provides a graphical user interface to support administration.

Usually the Lotus Notes Server is accessed over a local area network or the internet using a special client – the Lotus Notes Client. The *Lotus Notes client* is a graphical interface controlling the communication between the users and the system. A word processor for editing documents is integrated as well as a formula language to automatise tasks that occur repeatedly.

Recently, the Lotus Notes Server has been opened for web standards, which in particular means that it can be accessed over the internet protocol HTTP. When invoked over this protocol, the server dynamically generates HTML pages that are transferred over the internet and can be displayed by any standard web-browser (such as Netscape Communicator or Microsoft Explorer). In this case the web-browser plays the role of the client and hence an explicit installation of the Lotus Notes Client is obsolete. The part of the Lotus Notes server which deals with web-requests is called *Lotus Domino* server.

Application development is supported by a separate tool called *Domino Designer*. It is used for the development of databases and allows to adjust

all Lotus Notes design elements: forms, views, pages, outlines, framesets and agents.

- **Forms**: Forms are used to determine the structure data is stored in. Data stored by a form is called a document and consists of a number of fields. This way forms determine how data is edited and displayed. Please note that the same document can be displayed by different forms, which of course leads to different presentations of the same document.
- **Views**: Views are lists of documents sorted by particular criteria. They help users to find the documents they want and serve as summaries of the database contents. Every database must have at least one view, although most databases have more than one view.
- **Pages**: Pages make it possible to display text documents without having to define special forms. Each database contains a folder with pages where static pages may be deposited.
- **Outlines**: Outlines are navigation elements similar to classical folder structures. They are totally flexible in that the developer may integrate views, folders, links to other databases or homepages. Outlines can be used to build navigation structures which help the users to oversee some aspects of an application.
- **Framesets**: Framesets are well-known from web applications. They are used to split the user interface into clearly defined subareas.
- **Agents**: Agents are used to automatise tasks (see above).

4.1.3 Security

Security in Lotus Notes is achieved by a bunch of coalescent tasks which are part of the security management: secure transmission of data, authentication, authorisation, certification, configuration, backups and protection against hardware failure.

Levels

The security tasks are distributed over a number of levels up from the network down to single fields. For each of these levels, which are listed in more detail below, the system administrator, database developer or database manager may protect the system against unwanted access.

- **Network**: Many companies use firewalls to protect their internal network and data against unwanted access. In order to permit external access to the Lotus Notes (or Lotus Domino) server by Lotus Notes

Clients or web-browsers the configuration of the network has to enable port 1352 or port 80, respectively. Any other port may be blocked.

- **Operating system**: The Lotus Notes server supports a number of server platforms: Windows NT, Windows 2000, OS/2, AIX, Linux, Sun Solaris, HP-UX, OS 390 and OS/400. Since Lotus Notes databases are stored as ordinary files on the operating system, the security of the operating system (including the underlying hardware) plays a vital role.

- **Servers**: An important step towards a secure system is the configuration of the server document. The administrator determines if anonymous access to the server is forbidden. To increase security, anonymous access ought to be denied. In case anonymous access is intended explicitely for a concrete database (e.g. for the home-page of a company) this database is to be replicated to a separate partition of the server where anonymous access is permitted. Since remote access to the server for the purpose of configuration is permitted, the access rights to the server document have to be set with great care – otherwise the core of the system may be opened unintentionally.

- **Databases, forms, views, documents and fields**: Security at these levels mainly consists of authorisation issues which we describe in detail in Section 4.2.

Responsibility

Responsibility is laid to a number of people or groups whose duty is to provide security at their respective levels. The following table gives an overview:

Level	Responsible
Network	network administrator
Operating system	operating system administrator
Server	server administrator
Database	database manager
Views	database developer
Forms	database developer
Document	author of the document
Section	author of the document
Field	database developer

Please note that in some cases the responsibility is limited to the respective level (as for the network administrator) whereas in other cases (as e.g. the database manager) it extends over all lower levels.

Transaction monitoring

Orthogonal to the above listed security levels is the mechanism of transaction monitoring. Its purpose is to keep track of all access and communication processes, and it is used to locate disturbances once they have occurred. All information is stored in a number of log–books, which (among others) monitor: certifications (by the server), actions of particular users, reading or writing access to databases, eMmail activities, usage of communication ports and server threads.

Authentication

Authentication is quite different for Lotus Notes Clients and for web-browsers – depending on the protocol used for communication between client and server.

Lotus Notes Clients use a proprietary communication and security protocol which is not applicable to web browsers. The user name and password are encrypted in a so–called *ID–file* which is generated by the server administrator and installed by the user on his or hers client. The ID–file also contains public and private key of the user.

Web-clients communicate with the server by means of the open HTTP-protocol, which entails some restrictions as far as Notes-specific security mechanisms are concerned. Authentication is achieved by *basic authentication* i.e. user name and password (the password is stored in the name and address book on the server). As HTML is a page-description language, security at the level of fields is not available for web-applications. When using web clients, Lotus Domino provides the wide spread SSL (secure sockets layer) security protocol which is based both on public and private keys.

Since the protocols used for authentication are so much different, there can not be a unified theory dealing with all protocols equally (see [2] for meta-theoretic investigations of authentication). Each protocol has to be investigated in detail in order to be certified as being secure. Indeed, there is a line of theoretical research in its own right dealing with the correctness of individual security protocols – e.g. see [33, 3]). Authentication is not in the scope of this thesis and hence taken for granted in the remainder.

So far we have dealt with all security mechanisms – except for authorisation. Since authorisation is the major focus of this thesis, we devote the treatment of authorisation in Lotus Notes a full section.

4.2 Authorisation in Lotus Notes

Authorisation, in contrast to authentication, is a concept which is implemented on the server and does not depend on any communication protocol used for communication between the server and the clients. In the process

of authentication the server gains certainty that the user operating from a particular client is the person he or she pretends to be. In the process of authorisation the server chooses – depending on the type of person (in particular depending on his or hers affiliations) – the information which is transmitted to the user or changed in databases, depending on the respective request.

The security concept of Lotus Notes is realized on several layers. This also applies to authorisation which is realized on the following levels: servers, databases, forms, views, documents and fields. Within the several layers Lotus Notes distinguishes between different kinds of access rights. On the level of databases, for example, Lotus Notes defines managers or developers which are equipped with extensive rights to modify documents but also readers which do not possess any means to modify documents. A more extensive description of the authorisation model implemented in Lotus Notes can be found in [6].

Prior to focusing on the levels in more details, we have to deal with the fundamental name and address book – i.e. the database in which to register all users authenticated for interaction.

4.2.1 Name and address book

All users have to be registered within a central database, which is called the *name and address book (NAB)*. Being at the heart of any Lotus Notes application, it is of great importance; access needs to be restricted to a small number of authorised administrators. The NAB is the central administration tool comprising documents for users, servers, configurations, groups, connections and many more. Being a database like any other, the NAB contains documents, ACLs, forms, views etc. (see below).

Each *personal document* describes the name, first name, user name, password, environment, email-server etc. for a particular user. For unambiguous identification, the user names have to be unique throughout the NAB. The HTTP password, which is used for authentication with web-clients, is encrypted before storage. Personal documents, which require at least user name and password, may be created by administrators or – if desired explicitly – by users themselves. Beyond that, personal documents may be created by agents, i.e. programs that are executed periodically or manually.

Group documents contain lists of users, servers or groups themselves. Usually, group documents are created by administrators or agents who determine the desired group structure. In case ordinary users (i.e. non-administrators) are permitted to modify group documents, this is achieved indirectly: these users may modify group documents in different databases which in turn are copied to the NAB by agents.

Once the first Notes server has been installed, Notes automatically creates a new NAB with the file name "names.nsf". In case additional servers

are added, this NAB gets replicated for each server.

4.2.2 Databases

The NAB is a database like any other – equipped with some special meaning. We will describe common features of Lotus Notes databases first and then see later how these features apply to the NAB.

Databases are containers collecting data (documents) that are edited by forms and stored in the file-system with the extension *.nsf*. A database "discussion bboard", for example, contains postings and responses (documents) that are edited by forms "Posting" and "Reply" respectively. Each document comprises a number of fields (e.g. author, title, date or body) of different types as e.g. text, date, rich text. Documents are accessed by views, i.e. lists of documents filtered and sorted by view-specific criteria.

The person who creates a new database is assigned manager-rights which means that he or she may perform any action on that database (defining new forms or views, editing documents, adjusting the access control list, ...). Each newly created database initially is empty. The manager implements the components (mainly forms and views) himself or delegates these tasks to database developers. It is possible to develop all components from the scratch, achieving highest flexibility, or to inherit components from database templates.

Generating a Lotus Notes application is kind of a boot-strapping process. The initiator of the application first installs a new Lotus Notes server. The system then creates a file *names.nsf* (NAB) and assigns manager-rights to the initiator. The initiator (manager) may generate new personal documents and thus is in the position to register users. By creating group documents the initiator may summarise people to different groups - a hierarchic group structure emerges. After creating new databases, the initiator may delegate manager-rights (just as other rights) to single people or groups of people which in turn start developing the database – the whole application emerges.

Access control list

Each database contains an *access control list (ACL)*, which controls the access rights users posses. Consequently, the ACL is a security mechanism of Lotus Notes which rules access and actions to the documents, components and the ACL itself. Single users, groups of users and servers may be listed in any ACL as long as they are registered in the NAB. Responsible for the administration of the ACL is the database manager (or the managers respectively) who may change the entries of the ACL.

Access rights for anonymous users are regulated by entry of the artificial person *Anonymous*. In order to prevent anonymous users from accessing the database, person Anonymous has to be excluded from any access rights.

Access rights

Access rights are classified along the lines of access levels which users or groups may achieve. In Lotus Notes there are seven predefined possibilities: managers, developers, editors, authors, readers, deliverers and no access.

- **managers**: Managers are equipped with the most far-reaching rights in this hierarchy. They are responsible for the whole database and hence may edit all documents, change the entries of the ACL or even delete the database. In particular, managers may perform any action other users of the database are entitled to.
- **developers**: Developers may change the structure of the database (fields, forms, views, agents, ...) and may perform any action that the remaining lower access levels are permitted, too.
- **editors**: Editors may create new documents and edit all documents – including documents created by other users. The purpose of editors is to administrate the contents of the database, that is the documents.
- **authors**: Authors may create new documents and edit or delete their own documents. They may not change any other document. Authors are intended to contribute documents to a database and to administrate these documents without interfering with other users.
- **readers**: Readers may only read documents. They may not create, edit or even delete any document.
- **deliverers**: Deliverers may create new documents but not change or read even their own documents. This access right e.g. is suitable for handing in exams.
- **no access**: Users that are assigned this status may not access the database at all. In security critical applications this is recommended as standard option.

The access rights a user or user group has, can be refined in the ACL by disabling certain standard tasks as creating documents or deleting documents.

Besides the above levels, access to forms, views, documents or even single fields can be controlled on a grain basis. In each case the access rights can be refined by restricting access to certain users or groups only.

Roles

Since refinement of access rights can be spread widely over the components, there is a means to structure and modularise the refinement: *roles*. Roles can be freely defined in the ACL of a database. A role summarises a set of users

or user groups that posses equal access rights with respect to a database. In a sense roles are similar to groups which also summarise users. The striking difference between groups and roles is that the central administrators assign users to groups in the NAB whereas the local manager of a database assigns users to roles in the ACL of the database.

In the implementation of a database the developers may use roles to enable or disable concrete access rights as reading or editing documents in certain cases. If users are added to or removed from roles, these changes can be implemented easily by reconfiguring the ACL. Furthermore, the manager of a database can detect all different kinds of access rights just by inspecting the ACL.

Of course, access control can also be achieved directly without usage of roles by "hard–encoding" names of users or groups. In case of any change occurring for the access rights, the whole development of the database would have to be updated. To this end, the manager would have to oversee the whole, extensive development of the database rather than the local and compact ACL.

4.2.3 Documents

Readers fields and *authors fields* are particular kinds of fields that control access on the level of documents (in addition to form access lists, which strictly speaking are no true security mechanism – see Section 4.2.5). If a document contains several authors or readers fields, their contents is summed up.

Readers fields

A readers field of a document explicitly lists the users who are permitted to read the document. As a consequence, a user cannot see a document in a view without reader access to the document.

Entries in a readers field cannot give a user more access rights than what is specified in the access control list (ACL) of the database; they can only further restrict access. Users who have been assigned "no access" to a database can never read a document, even if they are listed in a readers field. On the other hand, users with editor access or above in the ACL can be restricted from reading documents if they aren't included in a readers field.

Users who have editor (or higher) access to a particular database can read a document if

- they are listed in any readers field or authors field or
- the document contains no readers fields and no authors fields.

Authors fields

Authors fields work in conjunction with author access rights in the access control list (ACL) of the database. Assigning users author access in the ACL, the users can read documents in the database but cannot edit even their own documents. Listing users in an authors field expands access rights by allowing the listed users to edit the document.

Entries in an authors field cannot override the access control list of the database; they can only refine it. Users who have been assigned "no access" to a database can never edit a document, even if they are listed in an authors field. Users who already have editor (or higher) access to the database are not affected by an authors field (provided they are permitted to read the document). Authors fields affect only those users who have author access to the database. Please note that authors fields differ from readers fields at this point: readers fields may well affect users with editor access or above.

4.2.4 Fields

Access control on the levels of fields is realized by encryption. The encrypted contents of a field is only accessible to people who are in possession of the required key – all other users face an empty field. Since we are not dealing with encryption explicitly in this paper, we omit a detailed discussion of this issue.

4.2.5 Forms and views

If the developer wants only some users to see a view or to control access to the documents created from a form, the developer may create a view access list or form access list, respectively.

Creating view or form access lists hinder access but are no true security features. For this reason, we do not investigate these features in more detail.

4.3 The model

Above we have listed the design elements for Lotus applications: forms, views, pages, outlines, framesets and agents. Pages, outlines and framesets are of no relevance for authorisation – hence they are ignored in this section. Forms and views provide means to create, edit, display and summarise documents. In our framework Logos we need not provide special support for these design elements since they can just be seen as special operations. Agents are programs that are executed by servers. In Logos, agents are no more than operations. As far as access control is concerned, agents do not require any special treatment since servers may be included in the set of

subjects (and consequently server groups in the group hierarchy) and hence access control easily also extends to servers.

Lotus Notes applications consist of a numbers of databases – each of them realizes some aspect of the application. The core of any database are the documents.

4.3.1 Documents

In Lotus Notes documents are just collections of fields. Depending on the form used for creation, the fields may differ completely for different documents. In Logos, though, the documents have to be typed. We have decided to model documents as records consisting of all fields occurring in any form of the application. Of course, for concrete documents there will be a multitude of empty fields – depending on the respective form.

Modelling documents as co-records labelled with the names of the forms (as in Example 8) would not be adequate. Assume for example document x which was created by form F_1. Later it was edited by form F_2. If the fields of F_1 and F_2 were disjoint (at least for some parts) then the document would both contain fields from F_1 and F_2. No label of the co-record (i.e. no name of a form) would match exactly the new set of fields. The only way to model Lotus Notes documents with co-records was to postulate closedness of the co-record w.r.t all unions (the union of two forms would be a new, auxiliary form comprising the union of the fields of both forms). On the one hand, this closedness property would blow up the type of documents unbearably and on the other hand the relabelling of the sum constructors (for the co-record) was practically unmanageable. Our decision to have one single type for all documents amounts to using the union of all forms for each document – saving effort at the prize of "weakening" typing.

Lotus Notes predefines two fields that have to be shared by any document: $\$UpdatedBy$ and $form$. These two fields[1] are also the starting point for Lotus Notes documents in Logos. Documents additionally may contain readers and authors fields *readers* and *authors*. These fields are investigated in more detail in Section 4.3.5.

Definition 53 (Lotus Notes – Minimal fields)

$$
\begin{aligned}
&\textbf{\textit{record}}\ \mathit{lotus\text{-}notes}^{\,d} = \\
&\quad \$\mathit{UpdatedBy} : \mathit{subject}^{\mathit{lotus\text{-}notes}} \\
&\quad \mathit{form} : \mathit{string} \\
&\quad \mathit{readers} : \mathit{authors\text{-}readers} \\
&\quad \mathit{authors} : \mathit{authors\text{-}readers} \\
&\mathit{with}\ \ \mathit{authors\text{-}readers} = \{\mathit{subjects} : \mathit{subject}^{\mathit{lotus\text{-}notes}}\ \mathit{list}, \\
&\qquad\qquad\qquad\qquad \mathit{groups} : (\mathit{group\ Ref})\ \mathit{list}, \\
&\qquad\qquad\qquad\qquad \mathit{roles} : \mathit{role}^{\,d}\ \mathit{list}\}
\end{aligned}
$$

[1] Type $\mathit{subject}^{\mathit{lotus\text{-}notes}}$ is introduced in Section 4.3.2.

For any application, the types of the documents are extensions of type *lotus-notes*. Profile documents in Lotus Notes are ordinary documents as any other – hence they do not need any special treatment with respect to their types.

Documents can not be accessed directly in Lotus Notes. Forms are required to this end. Basically, a form determines the set of fields that can be edited. Any database d in Lotus Notes provides of a set of forms $F_1^d, \ldots, F_n^d$. Each form F_k^d comprises a number a_k of fields x_1^k, $x_{a_k}^k$ of type τ_1^k, $\tau_{a_k}^k$. As we have mentioned above, the documents in Logos collect fields from all forms.

Definition 54 (Lotus Notes applications – Documents)

$$\textbf{\textit{type}}\ \mathit{data}^d = \mathit{lotus\text{-}notes}^d + \{x_1^1 : \tau_1^1, \ldots, x_{a_1}^1 : \tau_{a_1}^1, \ldots, x_1^n : \tau_1^n, \ldots, x_{a_n}^n : \tau_{a_n}^n\}$$

There is one peculiarity, though, which is not captured by this equation and which has to be taken into account additionally for concrete applications. Assume field x occurring multiply in different forms. Such a field is listed only once in data^d. Its type is a sum type composed of all individual types of the field in the different forms. In case of multiple occurrences of the same type, the type is only listed once in the sum type.

Example 28 (Lotus Notes applications – Documents) *Recall Example 8 where we have defined two records main and response. These records can easily be translated to corresponding Lotus Notes forms. Any application that uses these two forms is modelled in Logos – as far as the documents are concerned – as follows.*

$$\begin{array}{l}\textbf{\textit{record}}\ \mathit{data}^{bboard} = \mathit{lotus\text{-}notes}^{bboard} + \\ \quad \mathit{author} : \mathit{subject}^{lotus\text{-}notes} \\ \quad \mathit{date} : \mathit{date} \\ \quad \mathit{categories} : \mathit{string\ list} \\ \quad \mathit{note} : \mathit{string} \\ \quad \mathit{parent} : \mathit{nat}\end{array}$$

Assume the types of field author were different for main and response: author : subject in main and author : group Ref in response. Field author then was assigned type $\{\!|\ \mathit{main} : \mathit{subject}, \mathit{response} : \mathit{group\ Ref}\ |\!\}$ in record data^{bboard}. In Lotus Notes fields may have different types – depending on the form that was used for the most recent storage of the document. This heterogeneity is represented in the fully typed framework Logos as a sum type.

When defining operations for the forms it will be helpful to have something like an "empty document" at hand. Empty document empty^{bboard} is a record of type data^{bboard} with value arbitrary for each individual field. □

4.3.2 Groups

In the previous example we have used type $subject^{lotus\text{-}notes}$ without explaining properly how subjects look like in our model of Lotus Notes. In Lotus Notes there are basically two kinds of subjects: users and servers. Both of them have to be registered in the public name and address book – to this end a document is stored for each of them which contains essential information for registration.

For users the names (first name and family name) are stored as well as the hierarchic and unique user name. As far as authentication is concerned, the ID of the user is saved for client access – analogously the HTTP-password is stored for web-access. Additional information as the location of the user's mail file is added. For servers the name, ID and a long list of further configuration data is kept. As far as authorisation is concerned, users and servers do not make much of a difference. All that is required is a unique name to identify the user or server, respectively.

Since we do not assume that the set of subjects if fixed for a particular application we might use any infinite set for the subjects. We have (arbitrarily) decided to represent subjects as strings (natural numbers e.g. would have been just as appropriate)[2]:

Definition 55 (Lotus Notes – Subjects)

$$\textbf{\textit{type}}\ subject^{lotus\text{-}notes} = string$$

As far as groups are concerned, Lotus Notes uses the same structure as Logos: Groups consist of members and subgroups. Additionally, groups may be administrated. We therefore can reuse Definition 13 for our model of Lotus Notes.

Definition 56 (Lotus Notes – Groups)

$$\textbf{\textit{type}}\ groups^{lotus\text{-}notes} = groups$$

4.3.3 Roles

In Section 4.2.2 we have mentioned the access levels available in Lotus Notes: managers, developers, editors, authors, readers, deliverers and no access. We will ignore the developers for Logos since this access level only controls access to the code of the implementation. We have decided for our applications, though, that the implementations are fixed in advanced already.

Please note that there is a distinction in Lotus Notes between access levels and roles (to be precise there is no connection between these two issues in Lotus Notes at all). In Logos, though, we are able to model access

[2]In the following we will write short "$subject$" for "$subject^{lotus\text{-}notes}$" to ease the presentation.

levels by roles – access levels as a construct in its own right hence is obsolete. Take for example access level author. We define a role author and adjust the guards for creating documents in the sense that authors may create documents. In case an author wants to modify a document we have to check if he or she is entered in the authors field.

Any Lotus Notes application that is modelled in Logos consequently has to contain at least above mentioned roles (i.e. access levels). There are some refinements of the access levels which we have ignored so far. For example, authors may be prevented from adding or removing documents. We restrict ourselves to the just mentioned cases which are the most important ones. For concrete applications other cases might as well be added – provided there was a need to do so.

Definition 57 (Lotus Notes – Roles)

$$\begin{aligned} \textbf{\textit{datatype}}\ role^{lotus\text{-}notes} = \ & Manager \mid Editor \mid Author \mid \\ & Author_{\neg add} \mid Author_{\neg remove} \mid Author_{\neg add, \neg remove} \mid \\ & Reader \mid Deliverer \mid NoAccess \end{aligned}$$

Beyond these roles, which are shared by all databases, each database may define its own set of roles. For the discussion bboards e.g. we will add a role *ProfileEditor* which determines the set of subjects that may edit the profile documents: $role^{bboard} = role^{lotus\text{-}notes} \mid ProfileEditor$.

4.3.4 Access control list

Access control lists in Logos consist of two functions: *acl-groups* mapping roles to sets of groups and *acl-roles* mapping roles to set of roles. The first function corresponds to what the manager may adjust in the ACL of a database in Lotus Notes. Hence there are no restrictions for this function in applications of Logos (except for the minimal requirement that there must always be some subject with manager rights – otherwise the ACL is stuck forever). The second function expresses dependencies between roles that are provided by the application. In case of Lotus Notes these dependencies are as explained in Example 10 (extended with cases for *Deliverer* and *NoAccess*).

Definition 58 (Lotus Notes – ACL)

$$\begin{aligned} & acl\text{-}roles^{lotus\text{-}notes} : role^{lotus\text{-}notes}\ ACL\text{-}roles \\ & acl\text{-}roles^{lotus\text{-}notes}\ Manager = [] \\ & \quad \mid\ Editor = [Manager] \\ & \quad \mid\ Author = [Editor] \\ & \quad \mid\ Author_{\neg add} = [Author] \\ & \quad \mid\ Author_{\neg remove} = [Author] \\ & \quad \mid\ Author_{\neg add, \neg remove} = [Author_{\neg add}, Author_{\neg remove}] \\ & \quad \mid\ Reader = [Author_{\neg add, \neg remove}] \\ & \quad \mid\ Deliverer = [Author_{\neg remove}] \\ & \quad \mid\ NoAccess = [Reader] \end{aligned}$$

Each application of Lotus Notes has to use this function for the ACL without change. This function may in particular not be updated while execution. Figure 4.1 depicts the resulting graph for relation *is-subrole*. Relation is almost (with exception of "*Deliverer*") a lattice. See [38] for lattice-based access control models.

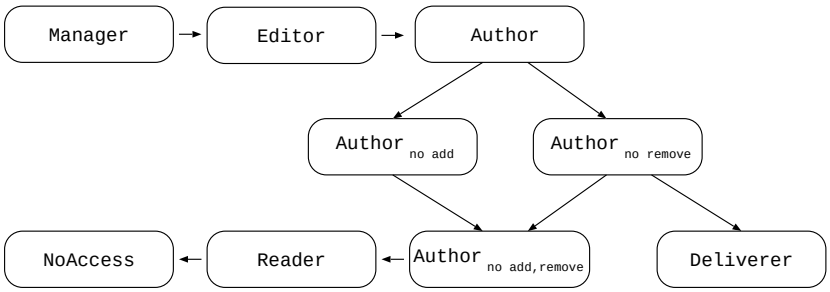


Figure 4.1: Is-subrole for Lotus Notes

Please note that *acl-roles* expresses positive consequences in the sense "if s is assigned role r_2 then s is also assigned r_1 – provided $r_2 \in$ *acl-roles* r_1". One might define a function *acl-roles*$'$ analogous to *acl-roles* which could express negative rights (see discussion in [41]) in the sense: "if subject s is assigned role r_2 then s must not be assigned role r_1 – provided $r_2 \in$ *acl-roles*$'$ r_1". With negative rights were were e.g. able to express that all subjects that are assigned role *NoAccess* may not be assigned any other role (in particular they might not be managers, which indeed is admissible in Lotus Notes).

To ease the presentation, we have abstained from negative rights for Logos, though they could easily be added. As Lotus Notes does not implement negative rights, the proposed framework Logos is sufficiently expressive for the authorisation model of Lotus Notes.

There is one little peculiarity, though, which we do not intend to model in Logos: in the ACL of any Lotus Notes database single users may be entered as well as groups consisting of single users only. Although from a theoretical point of view both of them are equivalent, Lotus Notes treats them differently. Single entries of users are dominant over all other entries of the same user in the sense that only the single entry is relevant to answer the question which access level the user is assigned to. Take for example user A which is listed personally in the ACL of a database and who is assigned *NoAccess*. No matter if A was member of a group assigned *Manager* level the user could not access the database. The situation was different if A was listed indirectly as the only member of a group. In this case the access level *Manager* would be dominant over *NoAccess* – hence user A would have manager rights.

Beyond what we have said so far, ACLs of Lotus Notes databases contain some speciality: default entries. The manager of a database uses this entry to specify some access level for default cases. This access level applies to a subject if and only if the subject (as member of a group) is not assigned any role explicitely in the ACL.

Some minor modifications of the Definitions in Chapter 3 suffice to account for default entries in Logos. The first thing to do is to store the access level assigned to the default entry. To do so, we extend the states of databases with an extra field for the default entry.

Definition 59 (Lotus Notes applications: States) *Assume d being the name of a database.*

$$\begin{aligned}\textbf{\textit{type}}\ \mathit{state}^d = \{\ &\mathit{data\text{-}space} : \mathit{data}^d\ \mathit{memory},\\ &\mathit{profile} : \mathit{profile}^d\ \mathit{memory},\\ &\mathit{acl\text{-}groups} : \mathit{role}^d\ \mathit{ACL\text{-}groups},\\ &\mathit{acl\text{-}default} : \mathit{role}^d\}\end{aligned}$$

The second modification concerns the lifting of guards to applications. Recall Definition 48 which first computes the assignees for all roles of the database and then calls the corresponding guard. The definition basically remains the same – only the function which computes the assignees takes the default case into account: if the subject under consideration is not assigned any role in the ACL, then it is added to the list of subjects assigned the role determined by the default entry.

Definition 60 (Lotus Notes applications: Lifting guards) *Assume any guard $P_h^{d_k}$ for internal or observable operation h^{d_k} of database d_k with following input types $(\mathit{group\ Ref} \to \mathit{subject\ list})$, $(\mathit{role}^{d_k} \to \mathit{subject\ list})$, $\mathit{subject}$, $\tau_1, \ldots, \tau_n$ and state^{d_k}. The lifted guard (lifted to the application a) is called $P_h^{a,\,d_k}$ and has type*

$$P_h^{a,\,d_k} : \mathit{subject} \to \tau_1 \to \ldots \to \tau_n \to \mathit{state}^a \to \mathit{bool}$$

It is defined as

$$\begin{aligned}&P_h^{a,\,d_k}\ s\ x_1\ \ldots\ x_n\ st =\\ &\quad \textbf{\textit{if}}\ \exists r.\ s \in g\ r\ \textbf{\textit{then}}\ P_h^{d_k}\ f\ g\ s\ x_1\ \ldots\ x_n\ st.st^{d_k}\\ &\qquad\qquad \textbf{\textit{else}}\ P_h^{d_k}\ f\ g[\mathit{default} := (g\ \mathit{default}) \cup \{s\}]\ s\ x_1\ \ldots\ x_n\ st.st^{d_k}\end{aligned}$$

with

$$\begin{aligned}&\mathit{acl} = \{\mathit{groups} = st.st^{d_k}.\mathit{acl\text{-}groups},\ \mathit{roles} = \mathit{acl\text{-}roles}^{d_k}\}\\ &f = \mathit{members}\ st.nab.\mathit{data\text{-}space}\\ &g = \mathit{assignees}\ st.nab.\mathit{data\text{-}space}\ \mathit{acl}\\ &\mathit{default} = st.st^{d_k}.\mathit{acl\text{-}default}\end{aligned}$$

4.3.5 Readers and authors fields

Readers and authors fields are a means to control authorisation on the level of single documents (see Section 4.2.3). In Lotus Notes each document may contain any number of readers or authors fields. The entries of the readers and authors fields respectively are added for authorisation. For our formal model we assume without loss of generality that there is exactly one readers and authors field. Furthermore, these fields are given fixed names: *readers* and *authors*.

Each readers or authors fields may contain a set of subjects, groups or roles. In Logos we model this as usually as a record.

Definition 61 (Readers and authors fields) *The readers and authors fields of any database d have following type*

$$\begin{aligned} \mathit{readers}^d,\ \mathit{authors}^d : \{&\mathit{subjects} : \mathit{subject}\ \mathit{list},\\ &\mathit{groups} : (\mathit{group}\ \mathit{Ref})\ \mathit{list},\\ &\mathit{roles} : \mathit{role}^d\ \mathit{list}\} \end{aligned}$$

In Lotus Notes forms need not necessarily contain readers or authors fields. If a form contains an empty readers or authors fields the field is treated as if it was non-existent. The special treatment of empty readers or authors fields gives rise to the convention in Logos that all documents must contain exactly one (possibly empty) readers and authors field. This justifies Definition 53 where we have taken readers and authors fields *readers* and *authors* to be constituent of any document.

Readers and authors fields are only meaningful at the time when the set of subjects is computed which is denoted by the entries of the field. Function *m-a* (*m-a* is short for *members-assignees*) serves this purpose.

Definition 62 (Meaning of readers and authors fields)

$$\begin{aligned} &\mathit{m\text{-}a} : (\mathit{group}\ \mathit{Ref} \to \mathit{subject}\ \mathit{list}) \to (\gamma \to \mathit{subject}\ \mathit{list}) \to\\ &\quad \{\mathit{subjects} : \mathit{subject}\ \mathit{list},\ \mathit{groups} : (\mathit{group}\ \mathit{Ref})\ \mathit{list},\ \mathit{roles} : \gamma\ \mathit{list}\} \to \mathit{subject}\ \mathit{list}\\ &\mathit{m\text{-}a}\ \mathit{nab}\ \mathit{acl}\ \{\mathit{subjects} = s,\ \mathit{groups} = g,\ \mathit{roles} = r\} =\\ &\qquad s \cup (\textstyle\bigcup_{x \in g} \mathit{nab}\ x) \cup (\bigcup_{x \in r} \mathit{acl}\ x) \end{aligned}$$

4.3.6 Operations and guards

Lotus Notes databases contain a number of standard operations to handle the documents and the ACL of the database. Precisely they are:

- **add-doc**: Adds documents to a database. In Lotus Notes forms are used to this end. For Logos we will allow any document (of proper type) to be added. If in concrete applications the restriction to particular forms is necessary, a set of functions for adding documents is required – each of them with the fields of the respective form as only input values.

- **map-doc**: Edits documents of a database. As for *add-doc* all fields may be edited. Restricting the update of documents to particular forms requires an analogous solution as for *add-doc*.

- **remove-doc**: Removes documents from a database. Only the fields of the documents are deleted. The references to the (empty) documents persist.

- **read-doc**: Displays documents of a database. In Lotus Notes views are usually used to this end (although single documents may be addressed by the unique document ID and hence displayed individually). Mimicking views in Logos again requires a set of functions – each of them with the columns of the views as output types.

- **map-acl**: Edits ACL of a database. In contrast with the documents there are no operations for creating or deleting ACLs.

Besides the operations for the documents of the dataspace there are analogous operations for the profile documents. Since there is an exact analogy between the operations for the data space and the operations for the profile documents, the operations for the profile documents are suppressed for the presentation.

In the following we define above listed operations and their corresponding guards. The guards are (together with the ACLs and the NAB) the heart of any Lotus Notes application – at least as far as authorisation is concerned.

The definition of the first operation *add-doc* is standard: a new object is allocated in the data space. Authors or above and deliverers are permitted in Lotus Notes to generate new documents – authors with the restriction that creation of documents must not be disabled for the authors. In Logos we have modelled the refinements of authors as roles $Author_{\neg add}$, $Author_{\neg remove}$ and $Author_{\neg add, \neg remove}$. Only the second of these roles is permitted to add new documents. If you look at Figure 4.1 you will find this to be captured precisely by the requirement that only deliverers and above are allowed to add new documents.

$$\begin{aligned}
&\mathit{add\text{-}doc} : \mathit{subject} \to \alpha \to (\alpha, \beta, \gamma)\mathit{state} \to (\alpha, \beta, \gamma)\mathit{state}\\
&\mathit{add\text{-}doc}\ s\ d\ st = st\{\mathit{data\text{-}space} := \mathit{alloc}\ d\ st.\mathit{data\text{-}space}\}
\end{aligned}$$

$$\begin{aligned}
&P_{\mathit{add\text{-}doc}} : (\mathit{group\ Ref} \to \mathit{subject\ list}) \to (\gamma \to \mathit{subject\ list}) \to\\
&\qquad \mathit{subject} \to \alpha \to (\alpha, \beta, \gamma)\mathit{state} \to \mathit{bool}\\
&P_{\mathit{add\text{-}doc}}\ \mathit{nab}\ \mathit{acl}\ s\ d\ st = s \in (\mathit{acl}\ \mathit{Deliverer})
\end{aligned}$$

Operation *map-doc* basically calls function *lift* which updates single objects in memories. The guard is a little bit more subtle. The first limitation is that only those documents are editable that are also visible. Clearly it makes little sense to edit invisible documents. Beyond that, authors fields come

into play. Please recall Section 4.2.2 where we have stated the following conditions for authors fields:

- Entries in an authors field can only refine the access control list of the database
- Users who already have editor (or higher) access to the database are not affected by an authors field (provided they are permitted to read the document)
- Authors fields affect only users who have author access to the database

The first condition is satisfied since the authors field only shows effect in conjunction with the requirement that the subject also must have at least $Author_{\neg add, \neg remove}$ access.

Users who have editor or higher access may edit any document (provided they are permitted to read it). This is expressed by the term $s \in acl\, Editor$. In this case the part of the disjunction which concerns the authors field is of no relevance since the disjunction always reduces to true.

As we have just shown, editors or above are not affected by authors fields. Subjects with access levels *NoAccess*, *Reader* or *Deliverer* must not edit any document – no matter if they are listed in the authors field. Subjects assigned any of these roles can not render the guard true as one can see easily from the definition.

$$
\begin{array}{l}
map\text{-}doc : subject \to (\alpha\, Ref) \to (\alpha \to \alpha) \to (\alpha, \beta, \gamma) state \to (\alpha, \beta, \gamma) state \\
map\text{-}doc\ s\ r\ f\ st = st\{data\text{-}space := lift\, f\ r\ st.data\text{-}space\}
\end{array}
$$

$$
\begin{array}{l}
P_{map\text{-}doc} : (group\ Ref \to subject\ list) \to (\gamma \to subject\ list) \to \\
\qquad subject \to (\alpha\, Ref) \to (\alpha \to \alpha) \to (\alpha, \beta, \gamma) state \to bool \\
P_{map\text{-}doc}\ nab\ acl\ s\ d\ st = P_{read\text{-}doc}\ nab\ acl\ s\ d\ st\ \wedge \\
\qquad (s \in acl\, Editor\ \vee\ (s \in acl\, Author_{\neg add, \neg remove} \wedge \\
\qquad\qquad s \in m\text{-}a\ nab\ acl\ (read\text{-}doc\ s\ d\ st).authors))
\end{array}
$$

Removing documents with operation *remove-doc* amounts to setting the value of the respective document to "*None*". The guard is analogous to the one for *map-doc* with the only exception that roles $Author_{\neg add, \neg remove}$ and $Author_{\neg remove}$ are excluded.

$$
\begin{array}{l}
remove\text{-}doc : subject \to (\alpha\, Ref) \to (\alpha, \beta, \gamma) state \to (\alpha, \beta, \gamma) state \\
remove\text{-}doc\ s\ r\ st = st\{data\text{-}space := lift\, (\lambda \_.\ None)\ r\ st.data\text{-}space\}
\end{array}
$$

$$
\begin{array}{l}
P_{remove\text{-}doc} : (group\ Ref \to subject\ list) \to (\gamma \to subject\ list) \to \\
\qquad subject \to (\alpha\, Ref) \to (\alpha, \beta, \gamma) state \to bool \\
P_{remove\text{-}doc}\ nab\ acl\ s\ d\ st = P_{read\text{-}doc}\ nab\ acl\ s\ d\ st\ \wedge \\
\qquad (s \in acl\, Editor\ \vee\ (s \in acl\, Author_{\neg add} \wedge \\
\qquad\qquad s \in m\text{-}a\ nab\ acl\ (read\text{-}doc\ s\ d\ st).authors))
\end{array}
$$

Operation *read-doc* dereferences a given reference in the data space. The

guard for this operation again is a little bit subtle since it involves readers fields. Please recall from Section 4.2.2 what we have said about readers fields:

- Entries in a readers field can only refine the access control list of the database
- Users who already have editor (or higher) access to the database can be restricted from reading documents if they aren't included in a readers field
- Editor (or higher) access to a database can read a document if it contains no readers and no authors field.

The first condition is satisfied since the readers field only occurs in conjunction with $s \in acl\ Reader$.

Users with editor (or higher) access are restricted from reading documents if the readers or the authors field is non-empty and the user is neither listed in the readers nor in the authors field.

If the document contains no readers and no authors field then the condition for the **if – then – else** statement computes to true. Since editors are higher in the hierarchy of access levels than readers, the term $s \in acl\ Reader$ allows editors to read the document.

$$
\begin{array}{l}
read\text{-}doc : subject \to (\alpha\ Ref) \to (\alpha, \beta, \gamma) state \to \alpha \\
read\text{-}doc\ s\ r\ st = dref\ r\ st.data\text{-}space
\end{array}
$$

$$
\begin{array}{l}
P_{read\text{-}doc} : (group\ Ref \to subject\ list) \to (\gamma \to subject\ list) \to \\
\qquad subject \to (\alpha\ Ref) \to (\alpha, \beta, \gamma) state \to bool \\
P_{read\text{-}doc}\ nab\ acl\ s\ d\ st = \\
\qquad \textbf{if}\ \ (read\text{-}doc\ s\ d\ st).readers = [] \ \wedge\ (read\text{-}doc\ s\ d\ st).authors = [] \\
\qquad\quad \textbf{then}\ \ s \in acl\ Reader \\
\qquad\quad \textbf{else}\ \ s \in acl\ Reader \ \wedge\ \ (s \in m\text{-}a\ nab\ acl\ (read\text{-}doc\ s\ d\ st).readers \vee \\
\qquad\qquad\qquad\qquad\qquad\quad s \in m\text{-}a\ nab\ acl\ (read\text{-}doc\ s\ d\ st).authors)
\end{array}
$$

The last operation *map-acl* is used to update ACL and default value at the same time. In case only one of them is intended to be modified, the identity function is used as update function for the other component. The guard of operation *map-acl* is quite trivial: only managers may change the ACL and the default value of a database.

$$
\begin{array}{l}
map\text{-}acl : subject \to (\gamma\ ACL\text{-}groups \to \gamma\ ACL\text{-}groups) \to (\gamma \to \gamma) \to \\
\qquad (\alpha, \beta, \gamma) state \to (\alpha, \beta, \gamma) state \\
map\text{-}acl\ s\ f\ g\ st = st\{acl\text{-}groups := f\ st.acl\text{-}groups\}\{acl\text{-}default := g\ st.acl\text{-}default\}
\end{array}
$$

$$
\begin{array}{l}
P_{map\text{-}acl} : (group\ Ref \to subject\ list) \to (\gamma \to subject\ list) \to \\
\qquad subject \to (\gamma\ ACL\text{-}groups \to \gamma\ ACL\text{-}groups) \to \\
\qquad (\gamma \to \gamma) \to (\alpha, \beta, \gamma) state \to bool \\
P_{map\text{-}acl}\ nab\ acl\ s\ f\ st = s \in (acl\ Manager)
\end{array}
$$

4.3.7 Name and address book

The name and address books of Lotus Notes applications in Logos do not differ much from what we have said in Section 3.2.7. NABs are ordinary databases with groups as data (to be precise administrated groups). The only subtlety concerns the administrators of administrated groups: they are permitted to change the group they are assigned to. In Logos we model this by requiring that the administrator groups have to be listed in the authors field of the corresponding document. Furthermore, all potential administrator groups have to be assigned role $Author_{\neg add}$ in the ACL of the NAB – i.e. administrators may change and remove their groups but not create new ones.

Chapter 5

Case study: Shared discussion bboards

In the previous sections we have defined the general framework Logos and have shown how to model the key authorisation concept of the groupware system Lotus Notes within this framework. It remains to show how concrete applications of this model look like. To this end we develop in this chapter a case study "Shared discussion bboards" which is based on the model of Lotus Notes developed in the previous chapter.

5.1 About

Discussion bboards are a wide-spread application of groupware systems allowing short notices to be exchanged among a variety of locally dispersed people. People who may access such a discussion bboard are assigned author and reader rights i.e. they may read and post notes (original postings just as well as replies). Beyond authors and readers there are also editors who may change any note. Basically, the purpose of the editors is to care for the quality of the postings (e.g. entries in adequate categories). On top of the access hierarchy is the database manager who may additionally change the ACL.

For many purposes these discussion bboards are satisfactory and quite useful. In more complex project structures, though, simple discussion bboards are too weak. Imagine a number of cooperating institutions (teams) working on the same project. Assume they intend to use a discussion bboard for their communication. Soon it will turn out, that there is information that only concerns a subset of the teams. Even stronger, this information has to be kept secret within the members of this subset.

Heterogeneous model The standard solution is to install several discussion bboards – rather than a single one. For each of the mentioned subsets a

discussion bboard is installed. Figure 5.1 depicts a sample structure, as implemented for the project "Daidalos" which aims at constructing an intranet for the German National Scholarship Foundation.

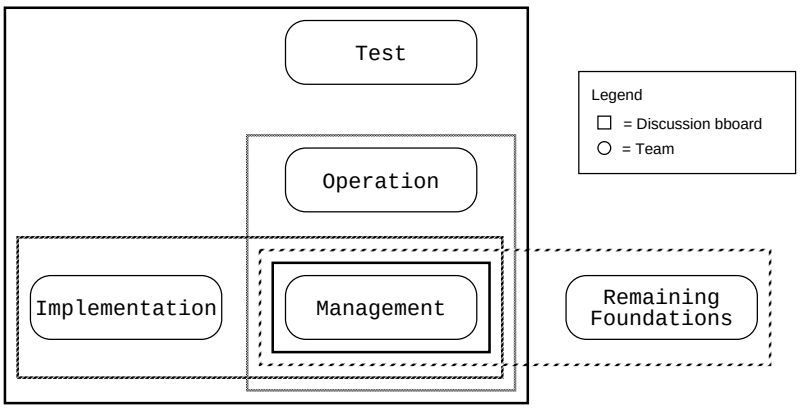


Figure 5.1: Structure of discussion bboards for "Daidalos"

The discussion bboard for the management team serves the collection of internal, project-relevant data. This data is meant for the management team only. Communication between the management and the implementors is kept track of by a second bboard. The operating team (responsible for graphics, contents, ...) also exchanges information with the management by means of a bboard. The test-users give feedback and are informed by a bboard which all of the just mentioned teams are subscribed to. Finally, information about the progress of the project is reported in a bboard to the remaining German scholarship foundations.

You can imagine that this multitude of bboards is source to permanent confusion. Take for example a note which is posted by the management team and which concerns both the operating team and the implementors. This note has to be duplicated – once for the implementors bboard and once for the operators bboard. If the note is edited later, it is quite likely to be edited only in one of the bboards rendering the contents of the two bboards inconsistent. Furthermore it is quite annoying for the management team to find the right bboard where a note has been posted. This structure makes it difficult for the management (and for other teams, too) to keep an overview over the project communication.

Homogeneous model The solution to the problem is – at least at first sight – simple: all participants share the same database (homogeneous model), access to documents is determined for each document individually. We call the database also "shared bboard".

For example, a manager of "Daidalos" could post a note and make it available to the implementors. Another note he could pass to the remaining scholarship foundations. In this homogeneous model the manager could see both notes in the same database without having to switch between different bboards as for the multitude of bboards. Even though this solution may seem easy from the users point of view, it has two major drawbacks: first, defining access for documents individually is orthogonal to the standard access model of Lotus Notes which defines access levels uniformly for databases and second, it complicates the administration of the bboard.

Of course, Lotus Notes is expressive enough to model access control for documents individually, but to some extend you lose the solid ground of ACLs having to use readers and authors fields throughout the application and thus having to guarantee the correctness of the implementation yourself.

In a heterogeneous model there are a number of ACLs (one for each bboard). In a sense this multitude of ACLs has to be mimicked in the single bboard of the homogeneous model. Take for example the editors. There is a group of editors for each individual bboard being able to read and edit all documents of the bboard. In the homogeneous model this set of editors has to be administrated in a single database – some new structure has to be introduced to this end. We will see in Section 5.2.1 that we even will have to provide a name and address book for each team individually.

You may wonder why we have chosen this example as case study for Logos. There are two good reasons for doing so. First, the problem stems from the real world where the homogeneous model could do good work for quite a number of projects (see Section 5.2). Second, and even more important, the example heavily extends the standard model (ACL-based access control) of Lotus Notes by making extensive usage of readers and authors fields. The correctness of such a complicated model is by no means obvious – hence there is a real need for a formal investigation of this security model.

In the next section we will show how the homogeneous model pays off for real-life projects (namely an intranet for all German scholarship foundations and a series of global student courses). Thereafter we will model the shared bboards in Logos and prove some properties.

5.2 Real world projects

The projects we describe in this section are slightly different in nature. The first project "An Intranet for the German Scholarship Foundations" has not been realized yet. Its security model (shared databases) is meant to be based on the same mechanisms as for the shared bboards. The case study of this chapter hence can serve as formal basis for this project. The second project

"A series of Global Student Projects" is running successfully already. Its infrastructure, which nowadays is based on the heterogeneous model, could benefit heavily from an implementation of the homogeneous model, though.

5.2.1 German Scholarship Foundations

The project is intended to be realized in two phases: in the first phase one intranet is built for each foundation taking project "Daidalos" as a blueprint. The project "Daidalos" (http://www.daidalos-projekt.de) currently implements an intranet for the German National Scholarship Foundation ("Studienstiftung des deutschen Volkes"). The key idea for all of the intranets is to connect the foundation's employees and the current and former scholars who are spread all over the world by an electronic network. More precisely, the internet is used for establishing connections, hence the word "intranet" does not imply installing physical wiring between participants. The word "intranet" denotes a logical subnet of the internet which is identified by personalised authorisation (password) mechanisms. Such subnets often are also called "virtual private networks".

In the second phase the separate intranets will be combined into one single intranet for all foundations. From the user's point of view this step will be invisible at first (a user of foundation x will only see a subnet of the whole intranet which is tailored for foundation x). The intranet of the second phase, though, will permit easy information exchange between the subnets of the foundations. You can compare the shift from phase one to phase two with a shift from the heterogeneous model to the homogeneous model for discussion bboards. In phase two of the project information exchange between users of different foundations can be regulated on a document basis. Assume for example a discussion bboard "Politics" which is shared between all foundations. Each author of a document then can decide which users (of which foundations) may read the document. Please note that each foundation has its own name and address book in the first phase and so needs to have one also in the second.

5.2.2 Global Student Projects

This section deals – in contrast with the former project, which has not been realized yet – with a running project (or to be precise a series of projects). Each of these projects is concerned with a realistic software engineering task which has to be implemented by students at the University of Technology in Munich and the Carnegie Mellon University in Pittsburgh collaboratively. A heterogeneous model of discussion bboards is used to this end. Since industrial partners are also involved in the projects, information can not be spread freely. In particular, the students have to sign a non-disclosure agreement which keeps information secret. Nevertheless, some part of the information

(as long as it does not underly the non-disclosure agreement) would also be relevant to subsequent projects. With a heterogeneous model information can only be passed with a copy and paste technique which is ineffective and leads to inconsistent data. A shared bboard could help to exchange information between different projects. Please note that each project has its own name and address book, hence the shared bboards have to respect the name and address books just as they had for the German scholarship foundations.

5.3 Realization in Lotus Notes

Shared discussion bboards provide a common (shared) dataspace for a set of teams. One of the key ideas is that each team may determine the access rights and register its members individually. Speaking in the terminology of Lotus Notes, each team needs its own name and address book. This is somehow in conflict with the requirement that all users (no matter in which of these NABs he or she is registered) share the same database: in Lotus Notes a database can only have one corresponding name and address book (even stronger, each domain has exactly one NAB).

Indeed, we will only have one central (global) name and address book, but its entries will be determined by the teams concurrently. Each team administrates its users and groups in a database (which we will call in the following a local NAB or the NAB of the team) which looks from the administrator's point of view like any NAB. Entries in this local NAB are of no direct relevance for the registration of users and groups in the whole system. These entries in the local NABs, though, are copied periodically by an agent (the so-called NAB agent) to the central NAB (which is a true NAB). Entries in the global NAB then are relevant for access control. Access to the global NAB is only granted to the central administrator and to the NAB agent which copies the entries from the local NABs to the global NAB.

Access control for the shared bboards is not achieved by entries in the ACL alone, but rather by a bunch of coalescent mechanisms. Take for example author rights that are required for those users who are permitted to discuss in a shared bboard.

The central administrator enters some group, say g, as authors in the ACL of the database (with the rights to create new and delete existing documents). Additionally, the administrator creates a new group g (which we will call a "shared group" since it has groups of different teams as subgroups) in the central NAB consisting exactly of subgroups $g_1 \ldots g_n$ (with n being the number of teams). Each team i determines the members of group g_i individually in its local name and address book NAB_i. The NAB agent copies for each team the corresponding subgroup to the central NAB.

Take for example the intranet for the German scholarship foundations

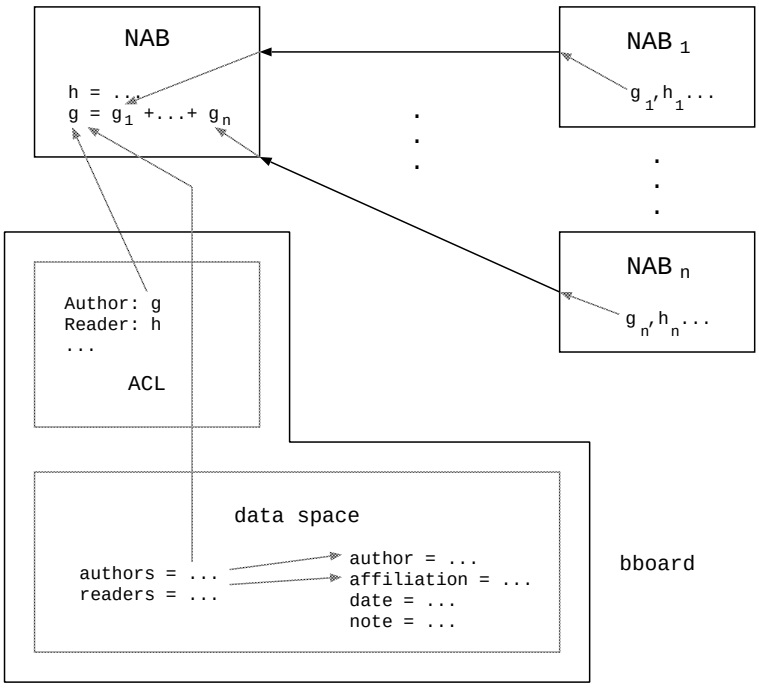


Figure 5.2: Shared bboards: access control

(see Section 5.2.1). Assume the foundations are sharing a bboard for discussions about science in general. Foundation f_1 e.g. lets all users take part in these discussions. It enters $g_1 = all_1$ in its NAB. Foundation f_2, in contrast, limits access to students only. Its entry in the local NAB looks like $g_2 = students_2$. By detour of the central NAB ($g = g_1 + \ldots + g_n$) these two entries are propagated to the ACL of the database where they take effect.

This way the central administrator can delegate clearly specified rights in a database to "third parties". The just mentioned mechanism is faithfully embedded in Lotus Notes in that it does not replace Lotus Notes specific elements like the ACL or roles – it uses them only in a non-standard way.

In the following we investigate in greater detail which "clearly specified" rights exist for shared discussion bboards and how to implement them in Lotus Notes. On the one hand we have to treat the standard access levels as *Reader* or *Author*, on the other hand we have to realize additional application specific roles as the editor of the categories.

- **Reader:** In ordinary databases the access level "Reader" determines who may read the documents. In our document based access model reading access has to be modelled by readers field. Since readers and authors field may not override but only refine the ACL of a database,

all potential readers have to be listed in the ACL. The central administrator generates some shared group – say g^1 – to this end. For each document there is a set of admissible teams which is selected by the author of the document. For each admissible team i group g_i^1 is added to the readers field.

- **Author:** The treatment of authors is similar to the readers. All potential authors have to be registered in the ACL by some shared group – say g^2. Since authors merely may edit their own documents, we store the names of the authors only in the authors field of their own documents.

- **Editor (global):** The global editors may edit (i.e. read, modify or delete) any document. The access level "Editor" of Lotus Notes almost serves this purpose. We add some group – say g^3 – (a shared group or a group from a concrete local NAB) to the ACL of the database and then we assign this group editor rights. This way the central administrator can delegate the selection of editors to single teams or to all teams – depending on whether the administrator chooses a shared group for g^3. Since editors are not affected by authors fields, we do not have to worry about the entries in the authors fields. In order to edit a document, the editor must be able to read it. To this end we add the whole group g^3 to any readers field.

- **Editor (local):** The local editors are like the global editors with the restriction that they may only edit documents of users affiliated with the same team. Listing the local editors as editors in the ACL certainly is wrong. Doing so would enable editors to edit any document they can read – and quite likely there will be documents of different teams released for reading, but not for editing. The local editors hence have to be modelled by authors and readers fields. The first thing to do is to introduce some shared group g^4. This group is given author access in the ACL which provides at least the potential to edit documents. For each document with author of team i group g_i^4 is added both to the readers and authors field of the document.

- **Categories editor:** The categories editor may change the list of predefined categories that are stored in the profile. For this role we introduce some group g_5 which we enter in the readers and authors field of the profile document. Additionally, we add group g_1 to the readers field, enabling readers of documents also to read the categories.

- **Manager:** None of the participating teams is superior to another. Therefore manager rights are reserved to a third party: the central administrator. The central administrator thus is the only person to

change the ACL of the database. Regardless of this right, the administrator should not modify the ACL while execution. Changes in the ACL (as performed for any other database) have been delegated for shared bboards concurrently to the teams.

So far we have assumed that the NAB agent respects the names of the groups (we have used the same names in the local NABs as well as in the global one). The names in the global NAB, though, have to be unique and they should unveil which team they stem from. One way to achieve this is by introducing some "meta-convention" encoding the team name in each name of a group. We have chosen this convention in Figure 5.2 encoding the name (number) of the team as subscript ($students_2$ e.g. are the students of foundation two). The NAB agent has to guarantee that each team sticks to the convention – it only copies those groups whose names are well-formed w.r.t. above convention.

A second way to achieve unique names was to refrain the local NABs from the name convention but enforcing the convention by the NAB agent, instead. In this case the names of the groups in the central NAB and in the local ones would diverge.

5.4 Modelling in Logos

For the representation in Logos (and also for an implementation in Lotus Notes) the second alternative is advantageous over the first. Since the name convention is guaranteed by construction of the NAB agent we avoid invariant conditions that state the well-formedness of the local NABs. Furthermore, the memory model for the local NABs is easier in the second than in the first case – as we shall see now.

5.4.1 References

Our previous definition of memories and references (see Definitions 5 and 6) implicitly assumes that multiple memories are independent i.e. there are no cross-references pointing to objects of foreign memories. In the central NAB, though, we have multiple memories (the local NABs) with cross-references (shared groups).
Of course we could simply paste the local NABs together in the NAB and merge the memories, but this would be awfully unmodular. As an example of *merging memories* see Figure 5.3 where we are merging the memories of NAB_1 and NAB_2. All references of NAB_2 would have to be redefined to point to the new memory cells in the merged memory. Even worse; assume some new reference was allocated for some local NAB (see e.g. Figure 5.4 where we have allocated a new object in the memory of NAB_1). The references in all subsequent NABs would have to be shifted by one. In

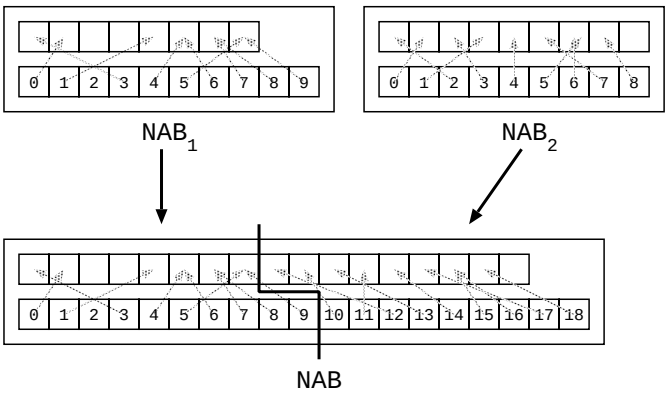


Figure 5.3: Merging memories

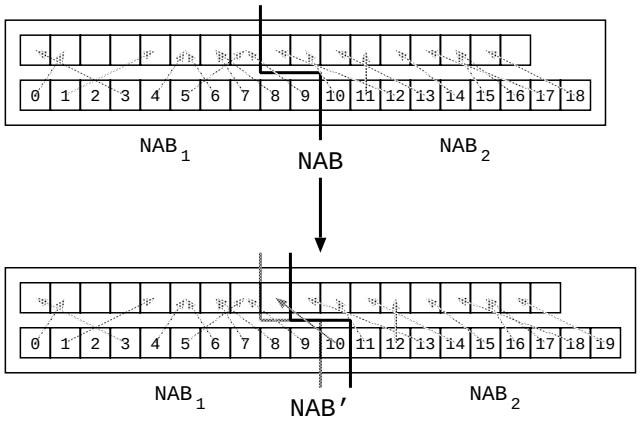


Figure 5.4: Merging memories is unmodular

other words: allocating a new object in one local NAB affects the whole application which is clearly in conflict with modularity.

Even though we could stick to our previous memory model and merge memories for the central NAB, such a decision would unnecessarily blow up the definitions and hence the correctness proofs. Instead, we choose a different memory model for the central NAB: products of memories, which we will call *composed memories*. The composed memory of the central NAB is a list of regular memories – one memory for each local NAB. This way both references and memories have to be redesigned.

Product references store pairs of natural numbers. The first natural number indicates the referenced (sub)memory in the memory list and the second natural number (e.g. see Figure 5.5) represents the reference locally within this submemory.

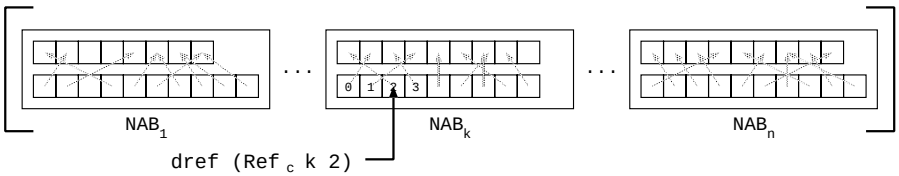


Figure 5.5: Dereferencing product references

The definitions of product references and composed memories now are straightforward.

Definition 63 (Product references)

$$\textbf{\textit{datatype}}\ \tau\ Ref_c = Ref_c\ nat\ nat$$

Definition 64 (Composed memories)

$$\textbf{\textit{type}}\ \ \tau\ memory_c = \{objs : \tau\ list,\ refs : nat\ list\}\ list$$

In Section 2.3 we have defined a general theory of references. To this end we have given a signature of operations in Definition 1 that have to be implemented by any memory model. In the following definition we realize the most important operations for composed memories and product references.

Definition 65 (Operations for composed memories)

$$\begin{array}{l}
new_{memory} : unit \to \tau\ memory_c \\
new_{memory}\ () = [\{objs = [],\ refs = []\}] \\
\\
alloc : \tau \to \tau\ memory_c \to \tau\ memory_c \\
alloc\ o\ m = update_{nth}\ 0\ \{objs = m'.objs{+}{+}o, \\
\qquad\qquad\qquad\qquad refs = m'.refs{+}{+}(length\ m'.objs)\}\ m \\
\quad with\ \ m' = nth\ 0\ m \\
\\
freshref : \tau\ memory_c \to \tau\ Ref_c \\
freshref\ m = Ref_c\ 0\ (length\ (nth\ 0\ m).refs) \\
\\
dref : \tau\ Ref_c \to \tau\ memory_c \to \tau\ option \\
dref\ (Ref_c\ n\ r)\ m\ \ =\ \ x := cell\ r'\ m'; \\
\qquad\qquad\qquad\qquad \textbf{\textit{if}}\ length\ m'.objs \leq x\ \textbf{\textit{then}}\ None \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{\textit{else}}\ Some\ (nth\ x\ m'.objs) \\
\qquad\qquad\qquad\qquad with\ \ r' = Ref\ r\ \ and\ \ m' = nth\ n\ m \\
lift : (\tau \to \tau) \to \tau\ Ref_c \to \tau\ memory_c \to \tau\ memory_c \\
lift\ f\ (Ref_c\ n\ r)\ m = \\
\quad update_{nth}\ n\ (m'\{objs := update_{nth}\ (cell\ r'\ m')\ (f\ (dref\ r'\ m'))\ m.objs\})\ m \\
with\ \ r' = Ref\ r\ \ and\ \ m' = nth\ n\ m \\
\\
store : \tau \to \tau\ Ref_c \to \tau\ memory_c \to \tau\ memory_c \\
store\ o = lift\ (\lambda\, x.\ o)
\end{array}$$

Please note that operations *alloc* and *freshref* only operate on the first memory (i.e. memory with index "zero"). This is adequate since all the user expects from these operations is a fresh memory cell – the user does not care which internal memory is used to this end.

5.4.2 Groups

With this renewed definition of memories and references, the groups and members require some minor, straightforward modification. Beforehand, we concretise the set of subjects for our application.

Subjects

As we have said already in Section 5.4.2, there are basically two kinds of subjects in Lotus Notes: users and servers. In the just mentioned section we have represented subjects (users or servers) as strings. For the NAB of shared bboards we follow this decision with one minor extension: we also store for users the team and for servers the domain as second component which we call the affiliation (*affil*). In the example of shared bboards we will only have one server "LocalServer" and one server domain "LocalDomainServers". As a convention, we store the local server at position zero of the affiliations. Central administrators are stored at the same position.

Definition 66 (Shared bboards: Subjects)

$$\textbf{\textit{type}}\ subject_c = \{name : string,\ affil : nat\}$$

The definition of groups follows immediately from the new definitions of subjects and references.

Definition 67 (Shared bboards: Groups)

$$\begin{aligned} \textbf{\textit{datatype}}\ group_c\ = \ & Empty\ | \\ & AddMember\ subject_c\ group_c\ | \\ & AddSubgroup\ (group_c\ Ref_c)\ group_c \end{aligned}$$

Members and Subgroups

What we have said in Section 3.1 applies directly to the shared memories and product references. For the definitions of members and subgroups we have only used the operations given in the signature of references (see Definition 1). Since the shared memories and product references instantiate this signature, we can be sure that the new memory model fits perfectly well in the existing structure. At this point we pick the fruits of our decision in Section 2.3 to generalise and define an abstract theory of references rather a concrete one.

5.4.3 Global and local NABs

As we have said, the access model for the shared bboards is non-standard and hence deserves formal investigation. There are two issues that make it non-standard. First, which we will treat in this section, is the distributed administration of the name and address book by means of an agent (the NAB agent). The second issue concerns the definition of access rights on the level of documents which requires some security critical agent (the access agent) to compute the authors and readers fields.

Data space

In the anteceding section we have sketched already how the memories (and thus the data spaces) for the local NABs and the global NAB look like: the local NABs are regular in the sense that they have ordinary memories as data spaces.

Definition 68 (Shared bboards: Data space for local NABs) *Index i denotes the i-th team.*

$$\textit{data-space}_i^{NAB} = \textit{group memory}$$

The global NAB, in contrast, uses the more sophisticated composed memory model with product references.

Definition 69 (Shared bboards: Data space for global NAB)

$$\textit{data-space}^{NAB} = \textit{group}_c\ \textit{memory}_c$$

Please note that the global NAB – in contrast with the local NABs – also uses product references for the subgroups.

NAB agent

The NAB agent operates on the data spaces of the local NABs. Precisely, it transforms the data spaces of a list of local NABs to the data space of the global NAB (see Figure 5.6).
There are two steps required for this transformation. First the agent transforms the list of regular memories into a composed memory and second the agent adds a set of shared groups to the composed memory. In our model these tasks are performed by two functions that are composed to complete the agent. Function *embed* is the first.

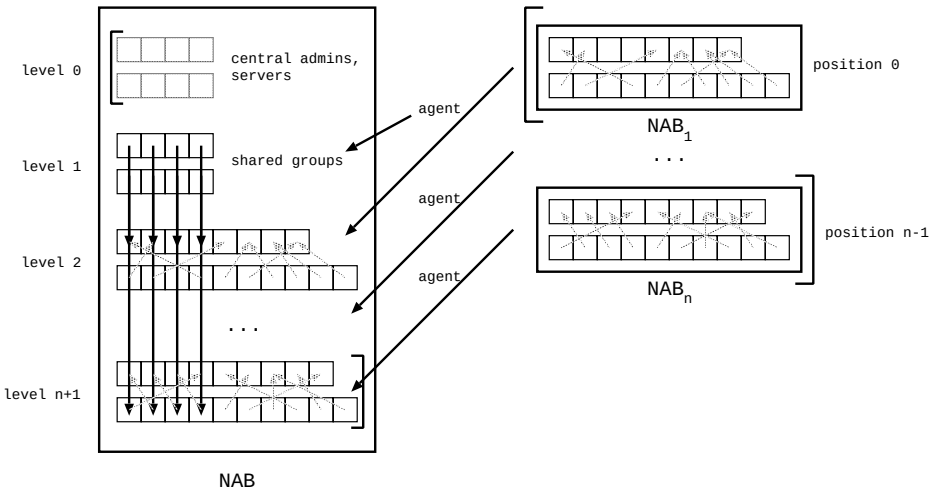


Figure 5.6: NAB agent

Definition 70 (Embedding: regular to composed memory)

$$
\begin{array}{l}
embed : (group\ memory)\ list \to group_c\ memory_c \\
embed = embed_{aux}\ 0 \\
\\
embed_{aux} : nat \to (group\ memory)\ list \to group_c\ memory_c \\
embed_{aux}\ n\ [] = [] \\
\quad | \quad (x \# xs) = x\{objs := map\ (embed_{groups}\ n + 2)\ x.objs\} \# \\
\qquad\qquad (embed_{aux}\ n\ xs) \\
embed_{groups} : nat \to group \to group_c \\
embed_{groups}\ n\ \ Empty \qquad\qquad\quad = Empty \\
\quad | \quad AddMember\ s\ g \qquad\quad = AddMember\ \{name = s,\ affil = n\}\ m \\
\quad | \quad AddSubgroup\ (Ref\ k)\ g\ = AddSubgroup\ (Ref_c\ n\ k)\ m \\
with\ m = embed_{groups}\ n\ g
\end{array}
$$

The auxiliary natural number introduced by function $embed_{aux}$ counts the position in the list of memories or the level in the composed memory, equivalently. Subjects are stored in the local NABs simply by names. The affiliation, which is also stored in the global NAB, is added by function $embed_{groups}$. Please note that the levels in the composed memory are shifted by two compared with the positions in the list of memories. This shift is performed to keep space for the central administrators (stored at level zero) and the shared groups (stored at level one). See Figure 5.6 for the levels of a NAB.

Example 29 (Embedding: regular to composed memory) *As an example we transform the regular memory of some local name and address book NAB_k (see Figure 5.7). Please note that regular and shared memories*

are accessed differently and that references to subgroups[1] *are adapted in the global NAB, too.*

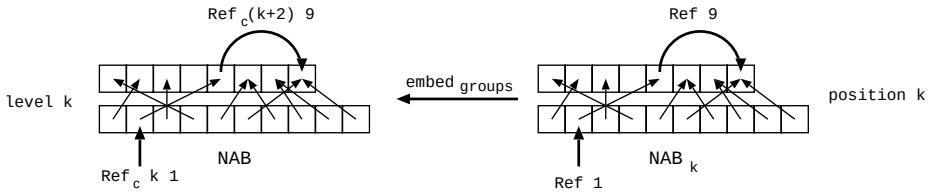


Figure 5.7: NAB agent – Embedding

Function *shared-groups* generates a composed memory with shared groups. It takes two natural numbers as input values. The first natural number determines the number of shared groups to be constructed. The second natural number gives the number of levels (except for the levels of the central administrators and the shared groups) which the composed memory consists of (or equivalently the length of the group memory list).

Definition 71 (Shared bboards: Generating shared groups)

$$
\begin{array}{l}
\mathit{shared\text{-}groups} : \mathit{nat} \to \mathit{nat} \to \mathit{group}_c\ \mathit{memory}_c \\
\mathit{shared\text{-}groups}\ n\ m = [\{\mathit{objs} = \mathit{shared\text{-}groups}_{\mathit{objs}}\ n\ m,\ \mathit{refs} = \mathit{shared\text{-}groups}_{\mathit{refs}}\ n\}] \\
\\
\mathit{shared\text{-}groups}_{\mathit{objs}} : \mathit{nat} \to \mathit{nat} \to \mathit{group}_c\ \mathit{list} \\
\mathit{shared\text{-}groups}_{\mathit{objs}}\ \mathit{0}\ m = [] \\
\qquad\qquad \mid \qquad (n+1)\ m = (\mathit{shared\text{-}groups}_{\mathit{objs}}\ n\ m) +\!+ \mathit{shared\text{-}group}\ (n+1)\ m \\
\quad \mathit{with} \\
\mathit{shared\text{-}group}\ n\ \mathit{0} = \mathit{Empty} \\
\qquad\qquad \mid \qquad (m+1) = \mathit{AddSubgroup}\ (\mathit{Ref}_c\ (m+3)\ n)\ (\mathit{shared\text{-}group}\ n\ m) \\
\\
\mathit{shared\text{-}groups}_{\mathit{refs}} : \mathit{nat} \to \mathit{nat\ list} \\
\mathit{shared\text{-}groups}_{\mathit{refs}}\ \mathit{0} = [] \\
\qquad\qquad \mid \qquad (n+1) = (\mathit{shared\text{-}groups}_{\mathit{refs}}\ n) +\!+ [n+1]
\end{array}
$$

Example 30 (Shared bboards: Generating shared groups) *In this example (see Figure 5.8) the composed memory of the NAB consists of two local NABs. The NABs share eight groups which are depicted at the top. These groups are generated by the term shared-groups 8 2.*

The NAB agent (to be precise its core; see Definition 75 for the complete definition of the NAB agent) is composed from functions *shared-groups* and

[1] In the picture the level of the memory in the shared group and the levels of references to subgroups diverge by two (level k but reference $\mathit{Ref}_c\,(k+2)\,9$). This is not by coincidence. When the central administrators and the shared groups are placed in front, all levels shift by two and the divergence vanishes.

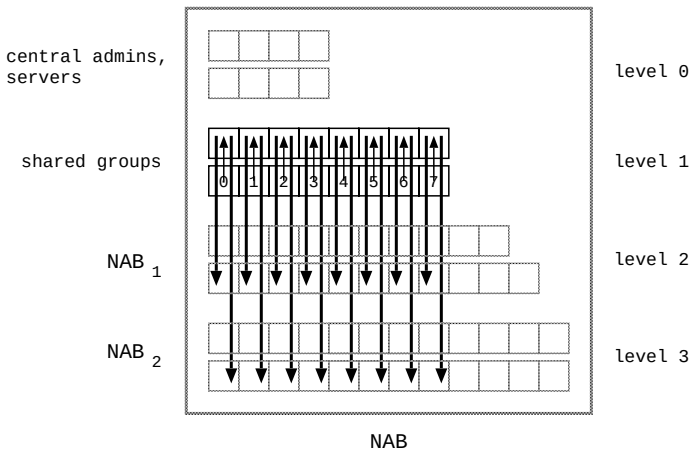


Figure 5.8: NAB agent – Generating shared groups

embed. It takes the number of shared groups as input and returns a memory transformer which transforms lists of regular memories to composed memories with shared groups.

Definition 72 (Shared bboards: NAB agent – core)

$$nab\text{-}agent_{core} : nat \to (group\ memory)\ list \to group_c\ memory_c$$
$$nab\text{-}agent_{core}\ n\ xs = (shared\text{-}groups\ n\ (length\ xs)) + + (embed\ xs)$$

Profile Documents

The local NABs do not contain any profile documents. For the global NAB there is only one profile document containing exactly one field: the number of shared groups. When executing the NAB agent, the server looks up this number in the profile document of the global NAB.

Roles

Both local and global NABs use the ordinary roles ("access levels") from Lotus Notes as defined in Definition 54.

Access control list

As a consequence from our decision in the last paragraph to take over the access levels of Lotus Notes without change, the ACLs of both local and global NABs are just standard (see Definition 58).

States

The major difference between local and global NABs is that each local NAB only stores one (regular) memory whereas global NABs store lists of memories (composed memory). Apart from that they only differ in the profiles which are empty for the local NABs and where the global NABs store the number of shared fields.

Definition 73 (Shared bboards: States of local NABs)

$$\begin{aligned}\textbf{\textit{type}}\ state^{NAB_{local}} = \{\ & data\text{-}space : group\ memory,\\ & profile : unit,\\ & acl\text{-}groups : role^{lotus\text{-}notes}\ ACL\text{-}groups,\\ & acl\text{-}default : role^{lotus\text{-}notes}\}\end{aligned}$$

Definition 74 (Shared bboards: States of global NABs)

$$\begin{aligned}\textbf{\textit{type}}\ state^{NAB} = \{\ & data\text{-}space : group_c\ memory_c,\\ & profile : nat,\\ & acl\text{-}groups : role^{lotus\text{-}notes}\ ACL\text{-}groups,\\ & acl\text{-}default : role^{lotus\text{-}notes}\}\end{aligned}$$

The NAB agent, finally, is a state transformer which takes the list of local NABs as input and transforms the list to the global NAB. Basically it applies $nab\text{-}agent_{core}$ to the list of memories of the local NABs. The number of shared groups to be generated is determined by the profile of the global NAB. The central administrators, which are stored at level zero, have to be respected (i.e. left unchanged) by the agent. Because the NAB agent is modelled in our formal model as an ordinary operation, it has an additional parameter for the subject that is permitted to invoke the operation.

Definition 75 (Shared bboards: NAB agent)

$$\begin{aligned}&nab\text{-}agent : subject_c \to state^{NAB_{local}}\ list \to state^{NAB} \to state^{NAB}\\ &nab\text{-}agent\ s\ xs\ st =\\ &\quad st\{data\text{-}space := (nth\ 0\ st.data\text{-}space) +\!+\\ &\qquad\qquad nab\text{-}agent_{core}\ (st.profile)\ (map\ data\text{-}space\ xs)\}\end{aligned}$$

It remains to show how initial states look like for local and global NABs. There are four issues worth mentioning:

- We take the number of shared groups as input value for the global NAB. This value is assumed fixed throughout the application although it could be kept variable (with some extra effort).
- The central administrator is "Admin" and the local server is "LocalServer" (they are the only subjects with administrator rights in the global NAB). These values are determined once and forever in the initial state as we do not provide any operation to change them (such operations could be added, complicating the case study unnecessarily).

- For the the local NABs we take the names of the managers as input. These values are subject to changes in the ACLs of the local NABs.
- We set the default values for both local and global NABs to *NoAccess* which excludes subjects not listed explicitely in the ACL from accessing the databases.

Definition 76 (Shared bboards: Global NAB – initial state)

$$
\begin{array}{l}
\mathit{initial\text{-}state}^{NAB} : \mathit{nat} \to \mathit{state}^{NAB} \\
\mathit{initial\text{-}state}^{NAB}\ n = \\
\quad \{\mathit{data\text{-}space} = \mathit{new}_{\mathit{memory}}; \\
\qquad\qquad \mathit{single\text{-}user}_{\mathit{group}}\,\{\mathit{name} = \text{“}\mathit{Admin}''\text{, } \mathit{affil} = 0\}; \\
\qquad\qquad \mathit{single\text{-}user}_{\mathit{group}}\,\{\mathit{name} = \text{“}\mathit{LocalServer}''\text{, } \mathit{affil} = 0\}, \\
\quad\ \mathit{profile} = n, \\
\quad\ \mathit{acl\text{-}groups} = \lambda\, x.\ \textbf{\textit{if}}\ x = \mathit{Manager}\ \textbf{\textit{then}}\ [\mathit{Ref}_{\,c}\ 0\ 0,\ \mathit{Ref}_{\,c}\ 0\ 1]\ \textbf{\textit{else}}\ []\} \\
\quad\ \mathit{acl\text{-}default} = \mathit{NoAccess}\}
\end{array}
$$

Definition 77 (Shared bboards: Initial states of local NABs)

$$
\begin{array}{l}
\mathit{initial\text{-}state}^{NAB_{local}} : \mathit{string} \to \mathit{state}^{NAB_{local}} \\
\mathit{initial\text{-}state}^{NAB_{local}}\ s = \\
\quad \{\mathit{data\text{-}space} = \mathit{single\text{-}user}_{\mathit{group}}\ s\ \mathit{new}_{\mathit{memory}}, \\
\quad\ \mathit{profile} = (), \\
\quad\ \mathit{acl\text{-}groups} = \lambda\, x.\ \textbf{\textit{if}}\ x = \mathit{Manager}\ \textbf{\textit{then}}\ [\mathit{Ref}\ 0]\ \textbf{\textit{else}}\ []\} \\
\quad\ \mathit{acl\text{-}default} = \mathit{NoAccess}\}
\end{array}
$$

Operations

The operations for the local NABs are just the standard operations from Lotus Notes (to be more precise from our model of Lotus Notes): *add-doc*, *map-doc*, *remove-doc*, *read-doc* and *map-acl* (see Section 4.3.6). There is only one operation for the global NAB namely the NAB agent *nab-agent*. The only way to register subjects in the global NAB is by the initial configuration (registration of the central administrator) and by execution of the NAB agent (registration of the subjects administrated in the local NABs).

Guards

Since we have used the standard operations of Lotus Notes for the local NABs, their guards (see Section 4.3.6) can be reused without change.

The situation is different for the global NAB. The only operation (the NAB agent) requires some investigation. This agent is the core of the whole application and hence may only be invoked by selected subjects. We assume that only managers of the global NAB (central administrator and local server) are trustworthy to this end. This is reflected by the definition of the guard for the NAB agent.

Definition 78 (Shared bboards: NAB agent – Guard)

$$
\begin{aligned}
&P_{nab\text{-}agent} : (group_c\ Ref_c \to subject_c\ list) \to \\
&\qquad (role^{lotus\text{-}notes} \to subject_c\ list) \to \\
&\qquad\quad subject_c \to state^{NAB_{local}}\ list \to state^{NAB} \to state^{NAB} \to bool \\
&P_{nab\text{-}agent}\ nab\ acl\ s\ xs\ st = s \in (acl\ Manager)
\end{aligned}
$$

5.4.4 Bboard

In Chapter 3 we have used discussion bboards as a running example. The case study of shared bboards in this chapter follows the lines of this example but takes multiple, independent teams into account who administrate their members autonomously. Nevertheless, much of what we have said in Chapter 3 is relevant for this case study.

Data space

The discussion bboards have (at least in principle) not been defined for any particular groupware system. In Example 28 we have adapted the bboards to Lotus Notes. For the shared bboards some minor modifications are required. Beyond the renewed definition of subjects for shared bboards the data space also stores the list of teams that are permitted to read the document. Furthermore, we omit fields *form* and $\$UpdatedBy$ since they are of no relevance for our case study.

Definition 79 (Shared bboards: Data space of bboards)

$$
\begin{aligned}
&\textbf{\textit{record}}\ data^{bboard} = \\
&\quad author : subject_c \qquad\qquad date : date \\
&\quad categories : string\ list \qquad\qquad note : string \\
&\quad teams : nat\ list \qquad\qquad parent : nat \\
&\quad readers : \{|\ subjects : subject_c\ list, \qquad authors : \{|\ subjects : subject_c\ list, \\
&\qquad\qquad groups : (group_c\ Ref_c)\ list, \qquad\qquad groups : (group_c\ Ref_c)\ list, \\
&\qquad\qquad roles : role^{bboard}\ list\ |\} \qquad\qquad roles : role^{bboard}\ list\ |\}
\end{aligned}
$$

Profile Documents

The profile document only consists of one field – the list of categories: $profile^{bboard} = string\ list$. To keep the case study simple and to abstain from irrelevant details we assume this list to be fixed once and for all at creation of the application. Clearly, operations could be added to modify this list.

Roles

Basically, this database uses the roles $role^{lotus\text{-}notes}$ from Lotus Notes. There is only one role called $Editor_{local}$ which we add: $role^{bboard} = role^{lotus\text{-}notes}$ |

$Editor_{local}$. This role is orthogonal to all other roles in the sense that it is not in relation *is-subrole* with any other role.

$$\begin{array}{lll} acl\text{-}roles^{bboard}\ role^{lotus\text{-}notes} & = & acl\text{-}roles^{lotus\text{-}notes}\ role^{lotus\text{-}notes} \\ \mid \qquad Editor_{local} & = & [] \end{array}$$

The purpose of role $Editor_{local}$ is to determine the local editors (see Section 5.3) of the shared bboard. Access level "categories editor" has been omitted for this case study (as already mentioned).

Access control list

Besides the dependencies between roles, the ACL stores the assignment of groups to roles. Although local NABs coexist with the global NAB, only the global NAB is relevant for registration and hence for the set of available groups. Please note that the manager of a shared database may freely decide which kind of group to choose (shared group or group of some particular team). If the manager chooses a shared group, then the members are determined by the teams concurrently. In case the manager selects some group of a particular team, then no other team is involved.

To keep the case study simple, we have negotiated to change the number of shared groups while execution. We provide one shared group per relevant case i.e. for the following three (see Definition 80): *Author*, *Reader*, $Editor_{local}$. All together we will reserve four shared groups (one extra shared group for position zero where we store the managers of the local NABs).

Usually, the manager of a database uses the ACL to control which subject may gain access to which kind of information. During execution, the ACL may change. For shared bboards the situation is different. The task of ruling over rights is delegated to the the managers of the local NABs who administrate the subjects by "remote control". The local NABs may change over time (in analogy to changes in the ACLs of ordinary databases) whereas the entries in the shared bboard (usually) remain unchanged. The following initial configuration hence is assumed to be fixed for the case study.

Definition 80 (Shared bboards: Initial ACL of bboards)

$$\begin{array}{lll} acl\text{-}groups^{bboard} : role^{bboard} & & ACL\text{-}groups \\ acl\text{-}groups^{bboard}\ Manager & = & [Ref_c\ 0\ 0] \\ \mid \qquad Editor & = & [Ref_c\ 0\ 0] \\ \mid \qquad Author & = & [Ref_c\ 1\ 1, Ref_c\ 1\ 3] \\ \mid \qquad Reader & = & [Ref_c\ 1\ 2] \\ \mid \qquad NoAccess & = & [] \\ \mid \qquad Editor_{local} & = & [Ref_c\ 1\ 3] \\ \mid \qquad - & = & [] \end{array}$$

Please note that the local editors are additionally entered as authors and thus implicitly also as readers. Otherwise it could not be guaranteed that

they were able to read and edit the documents they are responsible for. The managers and the editors are the only groups that are not shared between the teams. Only the central administrators may manage the shared bboard and edit all documents.

States

The definition of states for shared bboards ($state^{bboard}$) is an instantiation of Definition 59 taking the parameter d (the name of the database) to be *bboard*.

Since we do not provide any means to change the list of the categories in the profile, the initial states need to have this list as input. The definition of $initial\text{-}state^{bboard}$ is – apart from the categories – standard.

Definition 81 (Shared bboards: Initial states of bboards)

$$
\begin{aligned}
&initial\text{-}state^{bboard} : string\ list \to state^{bboard}\\
&initial\text{-}state^{bboard}\ s = \{\ data\text{-}space = new_{memory},\\
&\qquad\qquad\qquad\qquad profile = s,\\
&\qquad\qquad\qquad\qquad acl\text{-}groups = acl\text{-}groups^{bboard},\\
&\qquad\qquad\qquad\qquad acl\text{-}default = NoAccess\}
\end{aligned}
$$

Operations and guards

Basically, this database only uses the standard operations from Lotus Notes: *add-doc*, *map-doc*, *remove-doc* and *read-doc* (see Section 4.3.6) together with the corresponding guards. There are only two minor changes required: First, the guard for operation *add-doc* additionally guarantees that only those documents are added where field *author* really contains the name of the author. Second, function *map-doc* has to respect the author of the document i.e. must leave field *author* unaffected.

Please note that we have omitted one operation: *map-acl*. The sample correctness proof in Section 5.5 crucially depends upon a constant setup of the ACL. The ACL is configured properly in the initial state . Since we do not provide any operation to update the ACL, the well-formedness property remains invariant. Had we implemented operation *map-acl*, we had to guarantee well-formedness of the ACL by some semantic invariant.

Apart from the standard operations, the database also comprises an agent (the ACL agent) which we also model as an operation. This agent is the core of access control for shared bboards. The access rights are assigned granularly on a document basis (using readers and authors fields).

Definition 82 (Shared bboards: ACL agent)

$$
\begin{array}{l}
\mathit{acl\text{-}agent} : \mathit{subject}_c \to \mathit{state}^{\mathit{bboard}} \to \mathit{state}^{\mathit{bboard}} \\
\mathit{acl\text{-}agent}\ s\ st = st\{\mathit{data\text{-}space} := st.\mathit{data\text{-}space}\{\mathit{objs} := \\
\quad \mathit{map}(\lambda\ d.\ d\{\mathit{readers} := \\
\qquad \{\mathit{subjects} = [d.\mathit{author}], \\
\qquad\ \mathit{groups} = \mathit{Managers} +\!+ \mathit{Editors} +\!+ \\
\qquad\qquad (\mathit{de}_{\mathit{local}}\ d.\mathit{author.affil}) +\!+ (\mathit{da}\ d.\mathit{author.affil}) +\!+ \\
\qquad\qquad (\mathit{dr}\ d.\mathit{author.affil}) +\!+ (\mathit{join}\ (\mathit{map}\ \mathit{dr}\ d.\mathit{teams})), \\
\qquad\ \mathit{roles} = [] \} \\
\qquad \{\mathit{authors} := \{\mathit{subjects} = [d.\mathit{author}], \\
\qquad\qquad \mathit{groups} = \mathit{de}_{\mathit{local}}\ d.\mathit{author.affil}, \\
\qquad\qquad \mathit{roles} = [] \} \\
\quad\ st.\mathit{data\text{-}space.objs}\}\} \\
\mathit{with} \\
\quad \mathit{Managers} = st.\mathit{acl\text{-}groups}\ \mathit{Manager} \\
\quad \mathit{Editors} = st.\mathit{acl\text{-}groups}\ \mathit{Editor} \\
\quad \mathit{Editors}_{\mathit{local}} = st.\mathit{acl\text{-}groups}\ \mathit{Editor}_{\mathit{local}} \\
\quad \mathit{Authors} = st.\mathit{acl\text{-}groups}\ \mathit{Author} \\
\quad \mathit{Readers} = st.\mathit{acl\text{-}groups}\ \mathit{Reader} \\
\quad \mathit{decompose} : \mathit{nat} \to \mathit{group}_c\ \mathit{Ref}_c \to \mathit{group}_c\ \mathit{Ref}_c \\
\quad \mathit{decompose}\ x\ (\mathit{Ref}_c\ n\ m) = \textbf{\textit{if}}\ n = 1\ \textbf{\textit{then}}\ \mathit{Ref}_c\ n\ x\ \textbf{\textit{else}}\ \mathit{Ref}_c\ n\ m \\
\quad \mathit{de}_{\mathit{local}}\ n = \mathit{map}\ (\mathit{decompose}\ n)\ \mathit{Editors}_{\mathit{local}} \\
\quad \mathit{da}\ n = \mathit{map}\ (\mathit{decompose}\ n)\ \mathit{Authors} \\
\quad \mathit{dr}\ n = \mathit{map}\ (\mathit{decompose}\ n)\ \mathit{Readers}
\end{array}
$$

It is not immediate to see why this definition captures our intuition. In Section 5.5 we will therefore prove some key properties of this agent. For the moment we feel comfortable with informal arguments only.

- **Managers:** Even though managers are highest in the hierarchy of access levels, they are excluded from reading documents if they are not listed in the readers field (provided the readers field is non-empty). Therefore the managers have to be listed explicitely in all readers fields. For the authors fields the situation is different. Authors fields are only relevant for authors – managers may edit all documents regardless if they are listed in the authors field or not (see Section 4.2.3).

- **Editors:** The treatment of editors is analogous to the one for managers. All editors have to be listed explicitely in the readers fields but not in the authors fields.

- **Local Editors:** The local editors are in contrast with the editors no access level in Lotus Notes but only a role. For this reason, they have no access rights from the beginning. All access rights have to be given explicitely. The first step was to enter all local editors as readers and authors in the ACL (see Definition 80). This gives them the potential to read and edit documents – depending on their appearance in the

readers and authors fields. As the name suggest, the local editors may only edit those documents whose authors are affiliated the same team. To this end we use auxiliary function *decompose* which decomposes a shared group and selects the constituent which belongs to a particular team (in this case the team of the author of the document). The local editors of the author's team ("$de_{local}\ author.affil$") are entered both in the readers and authors fields.

- **Authors:** Authors may only edit their own documents. Hence there is no necessity to enter authors other than the author of the current document in the authors field. Authors dominate readers in Lotus Notes in the sense that all authors are also readers. Hence it might seem natural to enter all authors in the readers field. In the light of shared bboards, though, this would be too generous. Authors should only be permitted to read a document if the author of the document belongs to the same team ("$da\ author.affil$").

- **Readers:** Of course readers have no rights to edit documents (except they are also authors or above). Therefore they are not mentioned in the authors fields. Amongst all readers of a shared bboard there are two kinds of subjects that may read a particular document. The first kind comprises all readers of the same team as the author of the document ("$dr\ author.affil$"). The second kind is determined by field *teams* of the document. All readers of all teams listed in this field are also readers of the document an hence entered in the readers field ("$join\ (map\ dr\ d.teams)$").

- **Author of the document:** The author of any document may read his or hers own document. As a consequence, the author is listed both in the readers and the authors field of the document ("$d.author$").

The guard for the ACL agent is considerably simple: all central administrators or servers may invoke the agent. In other words, all groups at level zero in the NAB are permitted to do so.

Definition 83 (Shared bboards: ACL agent – guard)

$$
\begin{aligned}
P_{acl\text{-}agent} : & (group_c\ Ref_c \to subject_c\ list) \to \\
& (role^{lotus\text{-}notes} \to subject_c\ list) \to \\
& \quad subject_c \to state^{bboard} \to state^{bboard} \to bool \\
P_{acl\text{-}agent}\ & nab\ acl\ s\ st = (s.affil = 0)
\end{aligned}
$$

5.4.5 Application

So far we have defined all constituents of shared bboards. It's time now to put the pieces together.

States

The definition for the state of the application is along the lines of Definition 45 with the exception that the number of local NABs may vary and hence we have to use lists of local NABs rather than a fixed number of record fields.

Definition 84 (Shared bboards: States of application)
Assume shared to be the name of the whole application of shared bboard.

$$\begin{aligned} \textbf{\textit{type}}\ state^{shared} = \{ & nab : state^{NAB}, \\ & local\text{-}nabs : state^{NAB_{local}}\ list, \\ & bboard : state^{bboard}\} \end{aligned}$$

The list of categories is the fist argument of function $initial\text{-}state^{shared}$. The number of local NABs is determined by the list of managers (second argument of the function). There is exactly one local NAB per manager. In the NAB we generate a fixed number ("4") of shared groups.

Definition 85 (Shared bboards: Initial states of application)

$$\begin{aligned} & initial\text{-}state^{shared} : string\ list \rightarrow string\ list \rightarrow state^{shared} \\ & initial\text{-}state^{shared}\ c\ m = \{ nab = initial\text{-}state^{NAB}\ 4, \\ & \qquad\qquad\qquad\qquad local\text{-}nabs = map\ initial\text{-}state^{NAB_{local}}\ m, \\ & \qquad\qquad\qquad\qquad bboard = initial\text{-}state^{bboard}\ c\} \end{aligned}$$

Operations

As far as the operations of the NAB and the bboard are concerned, there is almost no peculiarity. Basically, they can be treated just as any other operation of Logos along the lines of Definitions 46 and 47. The only exception concerns the NAB agent. When lifting the NAB agent to the application, the auxiliary parameter (list of local NABs) is dropped since its value is computed from the list of local NABs which is – in contrast to the level of the respective database – available at the level of the application.

For the operations of the local NABs the situation is slightly different. The lifting of operations (and also guards) assumes that there is exactly one field per database in the global state. Our decision to take lists of local NABs in the global state violates this assumption. To remedy this weakness we, have to use list update and list selection rather than field update and field selection for record types in the lifting of the operations.

Definition 86 (Local NABs: Lifting internal operations) *For some local NAB m assume internal operation* $f_i^{NAB_{local}} : subject \rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow state^{NAB_{local}} \rightarrow state^{NAB_{local}}$. *The corresponding lifted operation* $f_i^{shared, NAB_{local}}$ *is defined as follows:*

$$f_i^{shared,\,NAB_{local}} : subject_c \to \tau_1 \to \ldots \to \tau_n \to state^{shared} \to state^{shared}$$
$$f_i^{shared,\,NAB_{local}}\,\{name = s,\ affil = m\}\ x_1\ \ldots\ x_n\ st =$$
$$\quad lift^{NAB_{local}}\ m\ (f_i^{NAB_{local}}\ s\ x_1\ \ldots\ x_n)$$

with

$$\quad lift^{NAB_{local}} : nat \to (state^{NAB_{local}} \to state^{NAB_{local}}) \to (state^{NAB} \to state^{NAB})$$
$$\quad lift^{NAB_{local}}\ m\ f\ st = st\{local\text{-}nabs := update_{nth}\ m\ f\ st.local\text{-}nabs\}$$

Definition 87 (Local NABs: Lifting observational operations) *This definition is analogous to the lifting of internal operations, though using the following lifting function instead.*

$$lift^{NAB_{local}} : nat \to (state^{NAB_{local}} \to \tau_{n+1}) \to (state^{shared} \to \tau_{n+1})$$
$$lift^{NAB_{local}}\ m\ f\ st = f\ (nth\ m\ st.local\text{-}nabs)$$

Guards

In the last chapter we have shown how to lift guards to Lotus Notes applications (see Definition 60). This definition applies directly to the NAB and the bboard of shared bboards.

For the local NABs we have to refine this definition – in analogy to the lifting of operations – to account for list update and list selection rather than field update and field selection for record types. Furthermore, the guards have to guarantee that operations are only invoked for existing local NABs.

Definition 88 (Local NABs: Lifting guards) *For local NAB m and internal or observable operation $h^{NAB_{local}}$ assume guard $P_h^{NAB_{local}}$ with the following input types $(group_c\ Ref_c \to subject_c\ list)$, $(role^{NAB_{local}} \to subject_c\ list)$, $subject$, τ_1, ..., τ_n and $state^{NAB_{local}}$. The lifted guard (lifted to application shared) is called $P_h^{shared,\,NAB_{local}}$ and has type*

$$P_h^{shared,\,NAB_{local}} : subject_c \to \tau_1 \to \ldots \to \tau_n \to state^{shared} \to bool$$

It is defined as

$$P_h^{shared,\,NAB_{local}}\,\{name = s,\ affil = m\}\ x_1\ \ldots\ x_n\ st =$$
$$\quad (\textbf{\textit{if}}\ \exists r.\ s \in g\ r\ \textbf{\textit{then}}\ P_h^{NAB_{local}}\ f\ g\ s\ x_1\ \ldots\ x_n\ (nth\ m\ st.local\text{-}nabs)$$
$$\qquad\qquad \textbf{\textit{else}}\ P_h^{NAB_{local}} f\ g\{default := (g\ default) \cup \{s\}\}\ s\ x_1\ \ldots\ x_n$$
$$\qquad\qquad\qquad (nth\ m\ st.local\text{-}nabs))\quad \wedge$$
$$\quad (length\ (st.local\text{-}nabs) \leq m)$$

with

$$\quad acl = \{groups = (nth\ m\ st.local\text{-}nabs).acl\text{-}groups,\ roles = acl\text{-}roles^{lotus\text{-}notes}\}$$
$$\quad f = members\ st.nab.data\text{-}space$$
$$\quad g = assignees\ st.nab.data\text{-}space\ acl$$
$$\quad default = (nth\ m\ st.local\text{-}nabs).acl\text{-}default$$

Access control

Section 3.3.3, which guides the way to access control, also serves as road map for this section. The first step is to define the transition relation (i.e. the set of potential steps the system may take).

- **NAB:** We have defined only one operation for the global NAB and that is the NAB agent. Consequently, there is only one introduction rule for the NAB.
- **Local NABs:** For the local NABs we have fallen back upon the standard definitions of Lotus Notes and hence we may also use the standard introduction rules.
- **Bboard:** The bboard, finally, uses the standard operations from Lotus Notes in addition to the ACL agent. The only exception concerns (as already mentioned) operation *map-acl* which we have skipped.

With the operations and guards of the application in hand, the definition of the transition relation degenerates to a book keeping task.

Definition 89 (Shared bboards: Transition relation) *The inductive set* $transition^{shared} : (state^{shared} \times subject_c \times state^{shared})set$ *is defined by the following introduction rules (we write* $st \xrightarrow{s}_{shared} st'$ *rather than* $(st, s, st') \in transition^{shared}$*).*

Transitions for the global NAB *("appl" short for "shared, NAB"):*

$$\frac{P^{appl}_{nab\text{-}agent}\ s\ st}{st \xrightarrow{s}_{shared} nab\text{-}agent^{shared,\ NAB}\ s\ st}$$

Transitions for the local NABs *("appl" short for "shared,* NAB_{local}*"):*

$$\frac{P^{appl}_{add\text{-}doc}\ s\ d\ st}{st \xrightarrow{s}_{shared} add\text{-}doc^{appl}\ s\ d\ st} \qquad \frac{P^{appl}_{map\text{-}doc}\ s\ r\ f\ st}{st \xrightarrow{s}_{shared} map\text{-}doc^{appl}\ s\ r\ f\ st}$$

$$\frac{P^{appl}_{remove\text{-}doc}\ s\ r\ st}{st \xrightarrow{s}_{shared} remove\text{-}doc^{appl}\ s\ r\ st} \qquad \frac{P^{appl}_{map\text{-}acl}\ s\ f\ g\ st}{st \xrightarrow{s}_{shared} map\text{-}acl^{appl}\ s\ f\ g\ st}$$

Transitions for the bboard *("appl" short for "shared, bboard"):*

$$\frac{P^{appl}_{add\text{-}doc}\ s\ d\ st}{st \xrightarrow{s}_{shared} add\text{-}doc^{appl}\ s\ d\ st} \qquad \frac{P^{appl}_{map\text{-}doc}\ s\ r\ f\ st}{st \xrightarrow{s}_{shared} map\text{-}doc^{appl}\ s\ r\ f\ st}$$

$$\frac{P^{appl}_{remove\text{-}doc}\ s\ r\ st}{st \xrightarrow{s}_{shared} remove\text{-}doc^{appl}\ s\ r\ st} \qquad \frac{P^{appl}_{acl\text{-}agent}\ s\ st}{st \xrightarrow{s}_{shared} acl\text{-}agent^{appl}\ s\ st}$$

□

In the second step we define the set of admissible states. In Definition 85 we have determined the initial states of the application. This together with the just given transition relation leads to the set of admissible states.

Definition 90 (Shared bboards: Admissible states) *The set of admissible states* $admissible^{shared} : state^{shared}$ *set for the shared bboards is defined by the following introduction rules.*

$$initial\text{-}state^{shared}\ c\ m \in admissible^{shared}$$

$$\frac{st \in admissible^{shared} \quad \wedge \quad st \xrightarrow{s}_{shared} st'}{st' \in admissible^{shared}}$$

Definition 51 (admissible transitions) can be transferred easily.

5.5 Sample correctness proof

So far we have introduced three levels of models: framework Logos, Lotus Notes and the particular case study of shared bboards. In a sense, each level is a refinement of the antecedent. Many definitions were required to complete all the steps and hence the following question arises naturally: what are the definitions good for?

One of the targets we hit is "clear specifications" and thus robust software design. Code (or specifications) we write in our model is much more abstract than any code deployed for some application in a concrete groupware system. Hence from a conceptual point of view we have gained clarity which in itself is a honourable goal.

Beyond clear specifications we have also laid ground for verification of security critical aspects of groupware applications. The usefulness of Logos for this purpose, though, has not been shown yet in this thesis. To this end we formulate and prove some important aspect of shared bboards: local editors may edit and only edit all documents of authors with the same affiliation (we have stated this as informal definitional property for local editors in Section 5.3).

Of course there are many more interesting properties of shared bboards that are worth investigating. Our intention, though, is not to develop a complete theory of shared bboards but rather to demonstrate the proof methodology of Logos and to show that our framework is suitable to express and prove interesting properties. Since there are so many definitions (and thus also lemmas) involved in a comprehensive theory of shared bboards, this task is infeasible without computer support (namely theorem provers). Indeed, one could reasonably imagine as further work a reformulation of this thesis in Isabelle/HOL (see the concluding remarks in Section 6).

We will see in the following section that even for the selected property the number of definitions and lemmas is high and renders a manual proof

complex. Please note that this complexity does not stem from the profoundness of the theorem but from the number of definitions and lemmas involved. This is a general remark that does not only refer to our framework but rather is true for many application-oriented proofs.

5.5.1 Proof sketch

Now let's turn to the details of the envisaged correctness proof. Of course, the first step is to formulate the desired theorem. As already mentioned, it expresses that local editors may edit and only edit all documents of authors with the same affiliation.

$$
\begin{array}{l}
\forall st \in \mathit{admissible}^{shared}.\ \forall n \leq \mathit{length}\ st.\mathit{local\text{-}nabs}. \\
\forall x \in \mathit{members}\ (\mathit{nth}\ n\ st.\mathit{local\text{-}nabs}.\mathit{data\text{-}space})\ (\mathit{Ref}\ 3). \\
\quad (\mathit{read\text{-}doc}^{shared,\ bboard}\ \_\ r\ st).\mathit{author}.\mathit{affil} = m \ \wedge\ q.\mathit{affil} = q'.\mathit{affil} = 0 \ \Rightarrow \\
\qquad m = n + 2 \Rightarrow (st' \stackrel{s}{\Longrightarrow}_{shared} st'') \ \wedge \\
\qquad m \neq n + 2 \ \wedge\ st'' \neq st' \ \Rightarrow \neg(st' \stackrel{s}{\Longrightarrow}_{shared} st'')
\end{array}
$$

with

$$
\begin{array}{l}
s = \{\mathit{name} = x,\ \mathit{affil} = n + 2\} \\
st' = \mathit{nab\text{-}agent}^{shared,\ NAB}\ q\ (\mathit{acl\text{-}agent}^{shared,\ bboard}\ q'\ st) \\
st'' = \mathit{map\text{-}doc}^{shared,\ bboard}\ s\ r\ f\ st'
\end{array}
$$

The informal term "for all local editors" is represented by the universal quantification $\forall x \in \mathit{members}\ (\mathit{nth}\ n\ st.\mathit{local\text{-}nabs}.\mathit{data\text{-}space})\ (\mathit{Ref}\ 3)$. Reference *Ref 3* stores for each team the set of local editors. In the just mentioned term we therefore quantify over all local editors of team n (with n being any team). Depending on whether a local editor is $(n = m + 2)$ or is not $(n \neq m + 2)$ affiliated with the same team as the author of the document under consideration, the local editor may $(st' \stackrel{s}{\Longrightarrow}_{shared} st'')$ or may not $(\neg(st \stackrel{s}{\Longrightarrow}_{shared} st''))$ change the document.

Please note that the transition relation is semantic in the sense that it defines the set of possible transitions independent of the operation used. The just mentioned theorem hence also comprises all updates of the document that are provoked by operations other than $\mathit{map\text{-}doc}^{shared,\ bboard}$ (this is unimportant for the first case, but of great importance for the second).

In the second case, a local editor must not invoke $\mathit{map\text{-}doc}^{shared,\ bboard}$ – not even for the purpose of leaving the document unchanged (i.e. $f = id$). Nevertheless, the local editor may use any other operation (or none, which is expressed by the reflexivity of the transition relation – see Definition 49) to leave the document unchanged, hence the side condition $st' \neq st$.

In a second step we reduce the theorem to a lemma which looks pretty much the same as the theorem, only with the guard of *map-doc* exchanged for the transition relation.

$$\forall st \in admissible^{shared}.\ \forall n \leq length\ st.local\text{-}nabs.$$
$$\forall x \in members\ (nth\ n\ st.local\text{-}nabs.data\text{-}space)\ (Ref\ 3).$$
$$(read\text{-}doc^{shared,\ bboard}\ \_\ r\ st).author.affil = m \ \wedge\ q.affil = q'.affil = 0 \ \Rightarrow$$
$$m = n + 2 \Rightarrow P^{bboard}_{map\text{-}doc}\ s\ r\ f\ st' \ \wedge$$
$$m \neq n + 2 \Rightarrow \neg P^{bboard}_{map\text{-}doc}\ s\ r\ f\ st'$$

with

$$s = \{name = x,\ affil = n + 2\}$$
$$st' = nab\text{-}agent^{shared,\ NAB}\ q\ (acl\text{-}agent^{shared,\ bboard}\ q'\ st)$$
$$st'' = map\text{-}doc^{shared,\ bboard}\ s\ r\ f\ st'$$

This lemma is the core of the correctness proof and requires extensive analysis of the NAB agent, the ACL agent and the model of access control. It does not require direct induction over the set of admissible states, though. All induction that is needed for this lemma concerns an auxiliary lemma that expresses that the ACL entry for the local editors remains unchanged in the bboard.

Deriving the theorem from the lemma is easy in the first case. The introduction rule for $map\text{-}doc^{shared,\ bboard}$ in the transition relation immediately succeeds. For the second case the situation is different. If the guard does not hold for some operation then it need not be true that the transition relation does not hold for the operation. Assume for example a transition relation with two introduction rules

$$\frac{P\ st}{st \xrightarrow{s} f\ st} \qquad \frac{Q\ st}{st \xrightarrow{s} g\ st} \qquad \text{with } f\ st' = g\ st' \text{ and } P\ st' = true,\ Q\ st' = false \text{ for some } st'$$

From the fact that $Q\ st' = false$ one may not conclude that $st' \xrightarrow{s} g\ st'$ does not hold. Since $f\ st' = g\ st'$ the transition is equivalent to $st' \xrightarrow{s} f\ st'$ which in turn may be derived from the first introduction rule.

Nevertheless in this particular case, the "reverse" introduction rule holds for operation $map\text{-}doc^{shared,\ bboard}$ which can be established by induction over the definition of the transition relation.

$$st \overset{s}{\Longrightarrow}_{shared} st' \ \wedge\ st \neq st' \ \wedge\ st' = map\text{-}doc^{shared,\ bboard}\ s\ r\ f\ st \ \Rightarrow$$
$$P^{shared,\ bboard}_{map\text{-}doc}\ s\ r\ f\ st$$

This lemma finally conquers the second case of the theorem.

5.5.2 Proof realization

As we have said, the central lemma investigates the conditions under which the guard for operation $map\text{-}doc^{shared,\ bboard}$ holds. The lemma consists of two propositions: "local editors are correct" and "local editors are secure". With "local editor correct" we mean that local editors really may edit documents of authors with the same affiliation. The term "local editors secure" expresses that local editors may not edit any document of some author with a different affiliation.

Lemma 17 (Guard correct and secure for local editors)

$$
\begin{array}{l}
\forall st \in admissible^{shared}.\ \forall n \leq length\ st.local\text{-}nabs. \\
\forall x \in members\ (nth\ n\ st.local\text{-}nabs.data\text{-}space)\ (Ref\ 3). \\
\quad (read\text{-}doc^{shared,\,bboard}\_\ r\ st).author.affil = m\ \ \wedge\ \ q.affil = q'.affil = 0\ \Rightarrow \\
\qquad m = n+2 \Rightarrow P^{shared,\,bboard}_{map\text{-}doc}\ s\ r\ f\ st'\ \wedge \\
\qquad m \neq n+2\ \Rightarrow \neg P^{shared,\,bboard}_{map\text{-}doc}\ s\ r\ f\ st' \\
with \\
\quad s = \{name = x,\ affil = n+2\} \\
\quad st' = nab\text{-}agent^{shared,\,NAB}\ q\ (acl\text{-}agent^{shared,\,bboard}\ q'\ st) \\
\quad st'' = map\text{-}doc^{shared,\,bboard}\ s\ r\ f\ st'
\end{array}
$$

Proof: First we replace assumption $q.affil = q'.affil = 0$ with proposition $st' \in admissible^{shared}$. This is immediate by the introduction rules for $nab\text{-}agent^{shared,\,NAB}$ and $acl\text{-}agent^{shared,\,bboard}$ in the definition of the transition relation. Next, we expand the definition of $P^{shared,\,bboard}_{map\text{-}doc}$ and apply Lemma 22 (the ACL of bboards constantly equals $acl\text{-}groups^{bboard}$). As a consequence, both occurrences of $P^{shared,\,bboard}_{map\text{-}doc}$ are replaced by

$$
\begin{array}{l}
\textbf{if}\ \exists r'.\ s \in acl'\ r' \\
\quad \textbf{then}\ P^{bboard}_{map\text{-}doc}\ nab'\ acl'\ s\ r\ f\ st'.bboard \\
\quad \textbf{else}\ P^{bboard}_{map\text{-}doc}\ nab'\ acl'[NoAccess := (acl'\ NoAccess) \cup \{s\}]\ s\ r\ f\ st'.bboard \\
\text{with} \\
\quad acl = \{groups = acl\text{-}groups^{bboard},\ roles = acl\text{-}roles^{bboard}\} \\
\quad acl' = assignees\ st'.nab.data\text{-}space\ acl \\
\quad nab' = members\ st'.nab.data\text{-}space
\end{array}
$$

Lemma 19 (see below), which expresses that local editors are dealt with properly by the NAB agent, guarantees that there is a witness ($r' := Editor_{local}$) for the existentially quantified variable r'. In order to apply the lemma we need a trivial auxiliary lemma which shows that the ACL agent does not affect the local NABs (because of its simplicity we do not spell it out explicitely). After application of Lemma 19 (with instantiation $st := acl\text{-}agent^{shared,\,bboard}\ q'\ st$) the following goal remains.

$$
\begin{array}{l}
\forall st \in admissible^{shared}.\ \forall n \leq length\ st.local\text{-}nabs. \\
\forall x \in members\ (nth\ n\ st.local\text{-}nabs.data\text{-}space)\ (Ref\ 3). \\
\quad (read\text{-}doc^{shared,\,bboard}\_\ r\ st).author.affil = m\ \ \wedge\ \ st' \in admissible^{shared}\ \Rightarrow \\
\qquad m = n+2 \Rightarrow P^{bboard}_{map\text{-}doc}\ nab'\ acl'\ s\ r\ f\ st'.bboard\ \wedge \\
\qquad m \neq n+2\ \ \wedge\ \ st'' \neq st'\ \Rightarrow \neg P^{bboard}_{map\text{-}doc}\ nab'\ acl'\ s\ r\ f\ st'.bboard \\
\text{with} \\
\quad s = \{name = x,\ affil = n+2\} \\
\quad st' = nab\text{-}agent^{shared,\,NAB}\ q\ (acl\text{-}agent^{shared,\,bboard}\ q'\ st) \\
\quad st'' = map\text{-}doc^{shared,\,bboard}\ s\ r\ f\ st' \\
\quad acl = \{groups = acl\text{-}groups^{bboard},\ roles = acl\text{-}roles^{bboard}\} \\
\quad acl' = assignees\ st'.nab.data\text{-}space\ acl \\
\quad nab' = members\ st'.nab.data\text{-}space
\end{array}
$$

The next step is to expand the definition of $P^{bboard}_{map\text{-}doc}$. Since the definition is quite extensive and it occurs twice in two independent subgoals, we treat the two subgoals separately in the following.

Case 1: "$m = n + 2$"

This goal again is a conjunction and hence split for the proof into two independent subgoals. The first subgoal reduces by expansion of definition $P^{bboard}_{read\text{-}doc}$ to

$$\begin{array}{l}\textbf{if } d.readers = [] \wedge d.authors = [] \\ \quad \textbf{then } s \in acl'\ Reader \\ \quad \textbf{else } s \in acl'\ Reader \wedge (s \in m\text{-}a\ nab'\ acl'\ d.readers \vee \\ \qquad\qquad\qquad\qquad s \in m\text{-}a\ nab'\ acl'\ d.authors)\end{array}$$

with

$$d = read\text{-}doc^{bboard}\ s\ r\ st'.bboard$$

Regardless if the predicate for the conditional expression holds (and by the way it does not hold), this proposition is true by Lemma 19 ($s \in acl'\ Reader$) and Lemma 20 ($s \in m\text{-}a\ nab'\ acl'\ (read\text{-}doc^{bboard}\ s\ r\ st'.bboard).readers$ and $s \in m\text{-}a\ nab'\ acl'\ (read\text{-}doc^{bboard}\ s\ r\ st'.bboard).authors$).

The second subgoal equals (due to the definition of $P^{bboard}_{map\text{-}doc}$)

$$\begin{array}{l}s \in acl'\ Editor \vee (s \in acl'\ Author_{\neg add, \neg remove} \wedge \\ \qquad\qquad\qquad\qquad s \in m\text{-}a\ nab'\ acl'\ (read\text{-}doc^{bboard}\ s\ r\ st'.bboard).authors)\end{array}$$

The left part of the disjunction does not hold as local editors may not be ordinary editors. But this is irrelevant since according to Lemmas 19 and 20 both subterms of the conjunction are valid.

We have succeeded *Case 1: "$m = n + 2$"* now and turn to the next (last) case of the lemma.

Case 2: "$m \neq n + 2$"

Expanding the definition of $P^{bboard}_{map\text{-}doc}$ and using de Morgan's laws (to eliminate the leading negation) the goal rewrites to

$$\begin{array}{l}\neg P^{bboard}_{read\text{-}doc}\ nab'\ acl'\ s\ r\ st'.bboard \vee \\ (s \notin acl'\ Editor \wedge (s \notin acl'\ Author_{\neg add, \neg remove} \vee \\ \qquad\qquad\qquad\qquad s \notin m\text{-}a\ nab'\ acl'\ (read\text{-}doc^{bboard}\ s\ r\ st'.bboard).authors))\end{array}$$

It is irrelevant (and not certain either) if the subject is permitted to read the document because the right part of the outer disjunction holds. The conjunction in turn is validated by Lemma 19 ($s \notin acl'\ Editor$) and Lemma 20 ($s \notin m\text{-}a\ nab'\ acl'\ (read\text{-}doc^{shared,\ bboard}\ s\ r\ st').authors$). Now we are done with the second case and hence with the proof of the whole lemma. □

During the proof of the lemma we have hinted at some auxiliary lemmas (Lemmas 18 – 22). In the following we deliver the missing lemmas.

The first auxiliary lemma shows how groups of the local NABs are registered in the global NAB by the NAB agent. Each member x of some group *Ref* m affiliated with team n is registered as subject $\{name = x,\ affil = n + 2\}$ in group $Ref_c\ (n + 2)\ m$ of the global NAB. Furthermore, the registered subject is contained in the shared group $Ref_c\ 1\ m$ (provided $Ref_c\ 1\ m$ is a shared group). Instantiating $m := 3$ adjusts the lemma for the local editors (for this instantiation the precondition of the first proposition is met, as the third group is indeed a shared group).

Please note that this lemma is some fundamental lemma about the functionality of the NAB agent and does not require admissible states but only regular states as input. The lemma does not mention, which effect the NAB agent has on level zero of the global NAB.

Lemma 18 (NAB agent correct)

$$
\begin{array}{l}
\forall n \leq length\ st.local\text{-}nabs. \\
\forall x \in members\ (nth\ n\ st.local\text{-}nabs.data\text{-}space)\ (Ref\ m). \\
\quad m \leq st'.profile \Rightarrow\ s \in nab'\ (Ref_c\ 1\ m)\ \wedge \\
\quad s \in nab'\ (Ref_c\ (n + 2)\ m) \\
with \\
\quad s = \{name = x,\ affil = n + 2\} \\
\quad st' = nab\text{-}agent^{shared,\ NAB}\ q\ st \\
\quad nab' = members\ st'.nab.data\text{-}space
\end{array}
$$

Proof: The proof of the first proposition is based on the correctness of the second proposition and additionally requires some list induction over function *shared-groups* (see Definition 71). The second proposition is proved by list induction over function *embed* (see Definition 70). □

The second auxiliary lemma enumerates (some of the) roles that local editors are assigned in the bboard: $Editor_{local}$, $Author$, $Author_{\neg add, \neg remove}$ and $Reader$. Furthermore it lists one role that local editors certainly are not assigned to: $Editor$.

Lemma 19 (NAB agent correct and secure for local editors)

$$
\begin{array}{l}
\forall st \in admissible^{shared}.\ \forall n \leq length\ st.local\text{-}nabs. \\
\forall x \in members\ (nth\ n\ st.local\text{-}nabs.data\text{-}space)\ (Ref\ 3). \\
\quad s \in acl'\ Editor_{local}\ \wedge\ s \in acl'\ Author\ \wedge \\
\quad s \in acl'\ Author_{\neg add, \neg remove}\ \wedge\ s \in acl'\ Reader\ \wedge \\
\quad s \notin acl'\ Editor \\
with \\
\quad s = \{name = x,\ affil = n + 2\} \\
\quad st' = nab\text{-}agent^{shared,\ NAB}\ q\ st \\
\quad acl = \{groups = acl\text{-}groups^{bboard},\ roles = acl\text{-}roles^{bboard}\} \\
\quad acl' = assignees\ st'.nab.data\text{-}space\ acl
\end{array}
$$

Proof: All goals (except for $s \notin acl'\,Editor$) are simple consequences from the definition of $acl\text{-}groups^{bboard}$ (see Definition 80) together with Lemmas 18 and 22. The fact that the ACL of bboards is constant is also used for the last subgoal ($s \notin acl'\,Editor$). The lemma reduces the subgoal to $s \notin members\ st'.nab.data\text{-}space\ (Ref_c\ 0\ 0)$, which is proved by Lemma 21 falling back upon assumption $s.affil = n + 2$. □

The third auxiliary lemma describes some circumstances under which local editors are (or are not) listed in readers or authors fields of the bboard. For all documents with authors of the same affiliation, the local editors are contained both in the readers and authors fields. Local editors are not listed in the authors fields of documents that are owned by authors of a different team. As far as the readers fields of such documents are concerned, the local editors are only mentioned if their affiliation is included in the field *team* of the document. This case, though, is (in contrast to all other cases) not covered by the following lemma.

Lemma 20 (ACL agent correct and secure for local editors)

$$
\begin{array}{l}
\forall st \in admissible^{shared}.\ \forall n \leq length\ st.local\text{-}nabs. \\
\forall x \in members\ (nth\ n\ st.local\text{-}nabs.data\text{-}space)\ (Ref\ 3). \\
\quad (read\text{-}doc^{shared,\,bboard}\ \_\ r\ st).author.affil = m\ \Rightarrow \\
\qquad m = n + 2 \Rightarrow s \in m\text{-}a\ nab'\ acl'\ (read\text{-}doc^{shared,\,bboard}\ s\ r\ st').readers\ \wedge \\
\qquad m = n + 2 \Rightarrow s \in m\text{-}a\ nab'\ acl'\ (read\text{-}doc^{shared,\,bboard}\ s\ r\ st').authors\ \wedge \\
\qquad m \neq n + 2 \Rightarrow s \notin m\text{-}a\ nab'\ acl'\ (read\text{-}doc^{shared,\,bboard}\ s\ r\ st').authors \\
with \\
\quad s = \{name = x,\ affil = n + 2\} \\
\quad st' = nab\text{-}agent^{shared,\,NAB}\ q\ (acl\text{-}agent^{shared,\,bboard}\ q'\ st) \\
\quad acl = \{groups = acl\text{-}groups^{bboard},\ roles = acl\text{-}roles^{bboard}\} \\
\quad acl' = assignees\ st'.nab.data\text{-}space\ acl \\
\quad nab' = members\ st'.nab.data\text{-}space
\end{array}
$$

Proof: The proofs for all three subgoals are quite straightforward by first expanding the definition of *acl-agent* (see Definition 82) and then applying Lemma 18 together with Lemma 22. □

The fourth auxiliary lemma expresses that the NAB agent respects the levels of the NAB (except for level one): subjects may only be members of groups with the same affiliation as their own.

Lemma 21 (NAB well-formed)

$$
\begin{array}{l}
\forall st \in admissible^{shared}. \\
\quad \{name = s,\ affil = n\} \in members\ st.nab.data\text{-}space\ (Ref_c\ m\ \_) \Rightarrow \\
\qquad m = n\ \ \vee\ \ m = 1
\end{array}
$$

Proof: This lemma describes some property that is invariant for all admissible states. The proof is performed by induction over the inductively defined set *admissible*. Clearly, the proposition holds for the initial NAB. Since the NAB agent is the only way to change the NAB, the proof reduces to showing the invariance property for the NAB agent. The NAB agent, finally, does not update level zero, hence the induction hypothesis works for this case. Level one is trivial since it is the only exception of the proposition. All levels from level two onward are dealt with equally based on some auxiliary lemma for function *embed*: $0 \leq m \wedge \{name = s, affil = n\} \in members\,(nth\,m\,(embed\,xs))\,r \Rightarrow n = m + 2$. □

The fifth (and last) auxiliary lemma shows that the ACL of the bboard is fixed for admissible states. In particular it is frozen at the level of the initial ACL $acl\text{-}groups^{bboard}$.

Lemma 22 (ACL of bboards constant)

$$\forall st \in admissible^{shared}.\ st.bboard.acl\text{-}groups = acl\text{-}groups^{bboard}$$

Proof: This lemma again describes an invariant property for all admissible states. Earlier on we have decided to leave the ACL of the bboard unchanged throughout the application (since we have avoided any operation that might change the ACL) and hence we can prove that the ACL always coincides with the initial ACL. The proof is a simple induction over the inductive set *admissible*. The proposition clearly holds for the initial ACL and since no operation changes the ACL of the bboard, the proposition is invariant in all induction steps. □

We are almost done with the preparations for our main theorem. All that remains is an auxiliary lemma which shows the inversion of the induction rule for operation $map\text{-}doc^{shared,\,bboard}$.

Lemma 23 (Induction rule for $map\text{-}doc^{shared,\,bboard}$ – inversion)

$$st \overset{s}{\Longrightarrow}_{shared} st' \ \wedge\ st \neq st' \ \wedge\ st' = map\text{-}doc^{shared,\,bboard}\ s\ r\ f\ st \Rightarrow P^{shared,\,bboard}_{map\text{-}doc}\ s\ r\ f\ st$$

Proof: This lemma is proved by elimination (induction on the derivation) of the inductive relation $\Longrightarrow_{shared}$ in the assumptions.

Case 1: "$st' = nab\text{-}agent^{shared,\,NAB}\ s'\ st$*"*

From the different definitions of st' (assumption and case distinction) we may conclude that $map\text{-}doc^{shared,\,bboard}\ s\ r\ f\ st = nab\text{-}agent^{shared,\,NAB}\ s'\ st$. Operation $map\text{-}doc^{shared,\,bboard}$ changes the bboard whereas $nab\text{-}agent^{shared,\,NAB}$ only affects the NAB. Since bboard and NAB are disjoint, the just mentioned equation is a contradiction – of course only provided that operations

$\mathit{map\text{-}doc}^{\mathit{shared},\,\mathit{bboard}}$ and $\mathit{nab\text{-}agent}^{\mathit{shared},\,\mathit{NAB}}$ are not without effect, which is taken care of by assumption $st' \neq st$.

Cases 2 – 5: "Operations of the local NABs"

An analogous argument holds for the operations of the local NABs ($\mathit{add\text{-}doc}^{\mathit{shared},\,\mathit{NAB}_{\mathit{local}}}$, $\mathit{map\text{-}doc}^{\mathit{shared},\,\mathit{NAB}_{\mathit{local}}}$, $\mathit{remove\text{-}doc}^{\mathit{shared},\,\mathit{NAB}_{\mathit{local}}}$ and $\mathit{map\text{-}acl}^{\mathit{shared},\,\mathit{NAB}_{\mathit{local}}}$) since the operations of the local NABs only change the local NABs and the local NABs are disjoint from the bboard.

Cases 6 – 9: "Operations of the bboard"

Operations $\mathit{add\text{-}doc}^{\mathit{shared},\,\mathit{bboard}}$ and $\mathit{remove\text{-}doc}^{\mathit{shared},\,\mathit{bboard}}$ add or remove some document which operation $\mathit{map\text{-}doc}^{\mathit{shared},\,\mathit{NAB}_{\mathit{local}}}$ does not. As in the previous cases, this different behaviour leads to a contradiction. The ACL agent $\mathit{acl\text{-}agent}^{\mathit{shared},\,\mathit{bboard}}$ operates (among others) on the same document as $\mathit{map\text{-}doc}^{\mathit{shared},\,\mathit{NAB}_{\mathit{local}}}$ but the fields that are affected by these two operations are disjoint. The ACL agent only modifies the readers and authors fields which are left untouched by $\mathit{map\text{-}doc}^{\mathit{shared},\,\mathit{NAB}_{\mathit{local}}}$. Again, a contradiction is the consequence. The last operation $\mathit{map\text{-}doc}^{\mathit{shared},\,\mathit{NAB}_{\mathit{local}}}$, finally, is trivial by induction hypothesis. □

The desired theorem, which characterises access control for the local editors, is a simple consequence of the previous lemmas.

Theorem 3 (Access control for local editors correct and secure)

$$
\begin{array}{l}
\forall st \in \mathit{admissible}^{\mathit{shared}}.\ \forall n \leq \mathit{length}\ st.\mathit{local\text{-}nabs}. \\
\forall x \in \mathit{members}\ (\mathit{nth}\ n\ st.\mathit{local\text{-}nabs}.\mathit{data\text{-}space})\ (\mathit{Ref}\ 3). \\
(\mathit{read\text{-}doc}^{\mathit{shared},\,\mathit{bboard}}\ \_\ r\ st).\mathit{author}.\mathit{affil} = m \ \wedge\ q.\mathit{affil} = q'.\mathit{affil} = 0 \ \Rightarrow \\
\qquad m = n + 2 \Rightarrow st' \stackrel{s}{\Longrightarrow}_{\mathit{shared}} st'' \ \wedge \\
\qquad m \neq n + 2 \ \wedge\ st'' \neq st' \ \Rightarrow\ \neg(st' \stackrel{s}{\Longrightarrow}_{\mathit{shared}} st'')
\end{array}
$$

with

$$
\begin{array}{l}
s = \{\mathit{name} = x,\ \mathit{affil} = n + 2\} \\
st' = \mathit{nab\text{-}agent}^{\mathit{shared},\,\mathit{NAB}}\ q\ (\mathit{acl\text{-}agent}^{\mathit{shared},\,\mathit{bboard}}\ q'\ st) \\
st'' = \mathit{map\text{-}doc}^{\mathit{shared},\,\mathit{bboard}}\ s\ r\ f\ st'
\end{array}
$$

Proof: The first subgoal is immediate from Lemma 17 by the introduction rule for $\mathit{map\text{-}doc}^{\mathit{shared},\,\mathit{bboard}}$ in the definition of $\longrightarrow_{\mathit{shared}}$. The second subgoal also follows from Lemma 17 but with application of Lemma 23 rather than the introduction rule. **Q.E.D.**

5.5.3 Extensions

To ease presentation, we have omitted some of the operations. Now that we have finished the proof for the simplified case study, it is time to lean back and reconsider how an extension of the case study can be achieved without

compromising the theorem (at least substantially).

We have eliminated most of the operations for the global NAB. We would at least expect some operations to add new groups and some operations to remove or edit existing groups. There is one lemma, though, that limits the behaviour of these operations. Lemma 21 requires that all operations have to respect the levels of the NAB (except for level one) i.e. subjects may only be members of groups with the same affiliation as their own. As long as this property is satisfied, such operations may be added (of course such operations only make sense for level *0*, since all other levels are overridden by the NAB agent). Furthermore we have left out some operation to deal with the ACL of the global NAB. We recognise now, that such an operation would have no impact on the theorem and hence could be added without danger.

The local NABs are complete with respect to basic functionality. Hence there is no urgent need to introduce further operations. In the bboard we have incorporated all necessary operations for the documents, but we have left out some operation to change the ACL. Such an operation is problematic since we require in Lemma 22 that the ACL of the bboard remains unchanged during execution. For our simplified case study, this lemma was satisfactory – for an extended case study, though, it has to be refined. The proof of the theorem depends upon the following two properties of the ACL.

- The second subgoal of the theorem states that local editors may not edit any document with author of a different affiliation. Clearly, this proposition is only true as long as the local editors are not editors or managers of the database. For the simplified case study we have achieved this by entering group $Ref_c\ 0\ 0$ as initial managers and editors by and requiring that the ACL remains unchanged. For an extended case study you could either require that the editor and managers must have affiliation zero (in that case the theorem is not affected) or you could modify the second subgoal of the theorem: if a local editor may edit some document whose author is affiliated with a different team, then the local editor also must be editor of the database.

- The proof of the theorem crucially depends upon the fact that group $Ref_c\ 1\ 3$ is assigned both role $Author$ and role $Editor_{local}$. Please note that the concrete number *3* is irrelevant for the proof. Any natural number k would do as long as the level of the local group was replaced consistently both in the theorem and the ACL (and of course preserved during execution).

Further operations could be added at least in principle. It has to be investigated for each operation individually, though, if the operation affects the proof of the theorem.

Chapter 6

Conclusions

We have presented a logical framework (Logos) which can be used to express authorisation issues in groupware applications. Since our focus is on applications and not on the meta-theory of the systems themselves, we have used a shallow encoding in a standard higher-order logic called HOL. There are a number of theorem provers supporting HOL (e.g. Isabelle/HOL) – hence the development can also be used for a fully machine-checked verification of authorisation issues in security critical groupware applications.

To be honest, though, all definitions have been written in HOL, but they have not been implemented in a theorem prover yet. Experience has shown (e.g. see [25]) that this step requires additional, extensive effort – even if from a theoretical point of view no new hurdles have to been taken. In this thesis we have set out for the first challenge: a comprehensive definition of authorisation issues in HOL. The second challenge – a complete formulation in a theorem prover (in particular Isabelle/HOL) – remains as further work. We would like to stress that we believe that the work would be worth the effort – providing a solid and reusable basis for machine checked verification of authorisation issues in groupware applications.

The application of theorem provers for access control of groupware systems is new frontier. However, there is a long tradition of access control models in the context of databases (cf. [19, 8, 39, 37]) and also CSCW i.e. computer-supported concurrent work (cf. [40, 7, 43, 42]). In contrast with our approach, these models do not aim at verification of real-life applications in the first place.

Although we do have concrete applications in mind, we do not establish any formal relationship between our model Logos and the real-life applications. Establishing such a formal relationship is hopeless from the beginning. Real groupware systems as Lotus Notes have gained a complexity which eludes the systems from any (complete) formal investigation. Please note that such an approach is standard in engineering. Engineers build small models

of the world – often without understanding the whole picture. Nevertheless, these models are extremely useful in practice. Sometimes, the models have mirrored aspects of reality improperly or incompletely. In these cases the model is improved to account for the new requirements. We do the same for Logos. Experience (see Chapters 4 and 5) shows that Logos is suitable for a relevant part of the (groupware) world. Nevertheless, improvements or extensions are not excluded. Please note that our decision to use a shallow encoding rather than a deep one pays off exactly at this point. Since we use the well-established meta-theory of the underlying logic, we are free to perform any change without compromising the meta-theory. In deep embeddings, where we have a meta-theory tailored to the specific model, any change entails painful consequences for the meta-theory and hence the formal basis. In this sense we believe, that our light-weight shallow approach is a first step towards a proof-engineering of authorisation issues in groupware applications.

Bibliography

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706 – 734, 1993.

[2] M. Abadi, M. Burrows, and R. Needham. A logic of authentication. *Proceedings of the Royal Society*, 1871(426):233 – 271, 1989.

[3] G. Bella and L. C. Paulson. Mechanising ban kerberos by the inductive method. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification: CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 416 – 427. Springer-Verlag, 1998.

[4] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, 1940.

[5] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, and C. Muñoz. *The Coq Proof Assistant User's Guide, version 6.1.* INRIA-Rocquencourt et CNRS-ENS Lyon, 1996.

[6] L. D. Corporation. *Domino Designer: The Power to Build Secure Web Applications that Extend the Enterprise – Application Development with Domino Designer.* Lotus Development Corporation, 1999.

[7] P. Dewan and H. Shen. Flexible meta access-control for collaborative applications. In *Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work*, pages 247 – 256, 1998.

[8] R. S. Gail-Joon Ahn. Role-based authorization constraints specification. *ACM Transactions on Information and Systems Security*, 3(4), 2000.

[9] M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[10] R. Harper and B. C. Pierce. A record calculus based on symmetric concatenation. In *POPL 1991*, pages 131–142, 1991.

[11] J. Harrison. HOL done right. Unpublished draft, 1996.

[12] M. Hofmann, W. Naraschewski, M. Steffen, and T. Stroup. Inheritance of proofs. *Theory and Practice of Object Systems, Special Issue on Third Workshop on Foundations of Object-Oriented Languages (FOOL 3)*, 4(1):51–69, 1998.

[13] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *SIGPLAN*, 27(5), May 1992. Version 1.2.

[14] M. P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, University of Oxford, 1992.

[15] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1 –35, 1995.

[16] L. Lamport and L. C. Paulson. Should your specification language be typed? Technical Report 425, University of Cambridge Computer Laboratory, 1997.

[17] K. Läufer and M. Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.

[18] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, Sept. 1994.

[19] T. F. Lunt and E. B. Fernández. Database security. *Data Engineering Bulletin 13(4)*, 13(4):43–50, 1990.

[20] Z. Luo. Ecc: an extended calculus of constructions. In *Proc. of IEEE 4th Ann. Symp. on Logic in Computer Science (LICS'89)*, 1989.

[21] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1983.

[22] W. Naraschewski. *Object-Oriented Proof Principles using the Proof-Assistant Lego.* Diplomarbeit, Universität Erlangen, 1996.

[23] W. Naraschewski. Towards an object-oriented progification language. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 215–230. Springer-Verlag, 1997.

[24] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs: Intl. Workshop TYPES '96*, volume 1512 of *Lecture Notes in Computer Science*, pages 317–332. Springer-Verlag, 1998.

[25] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.

[26] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 349–366. Springer, 1998.

[27] T. Nipkow. *Isabelle/HOL. The Tutorial*, 1999. Unpublished. Available at `http://isabelle.in.tum.de/doc/tutorial.pdf`.

[28] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle's Logics: HOL*, 2000. http://isabelle.in.tum.de/doc/logics-HOL.pdf.

[29] T. Nipkow and C. Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[30] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*, 1996.

[31] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[32] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[33] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 1(6):85 – 128, 1998.

[34] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[35] A. Pitts. The HOL logic. In Gordon and Melham [9], pages 191–232.

[36] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[37] P. S. Ravi Sandhu. Access control: Principles and practice. *IEEE Communications*, 32(9), 1994.

[38] R. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11), 1993.

[39] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2), 1996.

[40] H. Shen and P. Dewan. Access control for collaborative environments. In *Proceedings of the ACM Conference on Computer–Supported Cooperative Work*, pages 51 – 58, 1992.

[41] K. Sikkel. A group-based authorization model for cooperative systems. In *Proc. European Conference on Computer-Supported Cooperative Work (ECSCW'97)*, pages 345 – 360. Kluwer, 1997.

[42] K. Sikkel and O. Stiemerling. User-oriented authorization in collaborative environments. In *Proc. 3rd International Conference on the Design of Cooperative Systems (COOP'98)*, volume 1, pages 175 – 183, 1998.

[43] H. ter Hofte. *Working Apart Together: Foundations for Component Groupware*. Telematica Instituut, 1998.

[44] P. Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 461 – 493, 1992.

[45] P. Wadler. Monads for functional programming. In *Marktoberdorf Summer School on Program Design Calculi*, volume 118. Springer-Verlag, 1992.

[46] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer-Verlag, 1997.

[47] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2000. http://isabelle.in.tum.de/doc/isar-ref.pdf.

[48] J. M. Wing. A symbiotic relationship between formal methods and security. In *Proceedings from Workshops on Computer Security, Fault Tolerance, and Software Assurance: From Needs to Solution*. CMU-CS-98-188, 1998.