An Access-Control Logic in HOL

SHIU-KAI CHIN
Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, New York 13244

August 2016

Preface

This manual is meant to be a companion to the textbook *Access Control, Security, and Trust: A Logical Approach*, [CO11], by Chin and Older. We have embedded the syntax and semantics of the access-control logic in [CO11] in the Cambridge Higher-Order Logic (HOL-4) theorem prover, [GM93]. Additionally, we have defined inference rules in HOL corresponding to the inference rules in [CO11].

Acknowledgments

Sue Older is co-author with Shiu-Kai Chin of the textbook *Access Control, Security, and Trust: A Logical Approach*, CRC Press, 2011, [CO11]. This textbook was developed in large part due to Air Force Research Laboratory's Advanced Course in Engineering Cybersecurity Boot Camp. Lockwood Morris, Ph.D., Syracuse University professor emeritus, provided much appreciated assistance in implementing the access-control logic in the Cambridge Higher-Order Logic (HOL-4) theorem prover, [GM93].

We are grateful to our students. They tested the access-control logic and its implementation in HOL as part of the Air Force Research Laboratory's Information Assurance Research Internship program and in Syracuse University's Cyber Engineering Semester, [OC12].

Contents

Chapter 1

Access-Control Logic: Syntax and Semantics

The access-control logic we use is fully described in *Access Control, Security, and Trust: A Logical Approach*, [CO11]. It is a multi-agent modal logic with Kripke semantics. We present an overview of the logic consisting of its syntax, semantics, and inference rules. This presentation is almost identical with our overview of the logic in [CMOV10].

1.1 Syntax

Principal Expressions Let P and Q range over a collection of principal expressions. Let A range over a countable set of simple principal names. The abstract syntax of principal expressions is:

$$P ::= A \mathrel{/} P\&Q \mathrel{/} P \mid Q$$

The principal $P\&Q$ ("P in conjunction with Q") is an abstract principal making exactly those statements made by both P and Q; $P \mid Q$ ("P quoting Q") is an abstract principal corresponding to principal P quoting principal Q.

Access Control Statements The abstract syntax of statements (ranged over by φ) is defined as follows, where P and Q range over principal expressions and p ranges over a countable set of *propositional variables*:

$$\begin{aligned}\varphi \quad ::= \quad & p \mathrel{/} \neg\varphi \mathrel{/} \varphi_1 \wedge \varphi_2 \mathrel{/} \varphi_1 \vee \varphi_2 \mathrel{/} \varphi_1 \supset \varphi_2 \mathrel{/} \varphi_1 \equiv \varphi_2 \mathrel{/} \\ & P \Rightarrow Q \mathrel{/} P \textsf{ says } \varphi \mathrel{/} P \textsf{ controls } \varphi \mathrel{/} P \textsf{ reps } Q \textsf{ on } \varphi\end{aligned}$$

Informally, a formula $P \Rightarrow Q$ (pronounced "P speaks for Q") indicates that *every* statement made by P can also be viewed as a statement from Q. A formula $P \textsf{ controls } \varphi$ is an abbreviation for the implication $(P \textsf{ says } \varphi) \supset \varphi$: in effect, P is a trusted authority with respect to the statement φ. $P \textsf{ reps } Q \textsf{ on } \varphi$ denotes that P is Q's delegate on φ; it is an abbreviation for $(P \textsf{ says } (Q \textsf{ says } \varphi)) \supset Q \textsf{ says } \varphi$. Notice that the definition of $P \textsf{ reps } Q \textsf{ on } \varphi$ is a special case of $\textsf{controls}$ and in effect asserts that P is a trusted authority with respect to Q saying φ.

1.2 Semantics

Kripke structures define the semantics of formulas.

Definition 1.1 *A* Kripke structure $\mathcal{M}$ *is a three-tuple* $\langle W, I, J\rangle$*, where:*

- W *is a nonempty set, whose elements are called* worlds.
- $I : \textbf{\textit{PropVar}} \rightarrow \mathcal{P}(W)$ *is an* interpretation *function that maps each propositional variable* p *to a set of worlds.*
- $J : \textbf{\textit{PName}} \rightarrow \mathcal{P}(W \times W)$ *is a function that maps each principal name* A *to a relation on worlds (i.e., a subset of* $W \times W$*).* ▪

We extend J to work over arbitrary *principal expressions* using set union and relational composition. The extended function $\hat{J}$ is defined as follows:

$$\begin{aligned}\hat{J}(A) &= J, \text{ where } A \text{ is a simple principal name}\\ \hat{J}(P\&Q) &= \hat{J}(P) \cup \hat{J}(Q)\\ \hat{J}(P \mid Q) &= \hat{J}(P) \circ \hat{J}(Q),\end{aligned}$$

where

$$\hat{J}(P) \circ \hat{J}(Q) = \{(w_1, w_2) \mid \exists w'.(w_1, w') \in \hat{J}(P) \text{ and } (w', w_2) \in \hat{J}(Q)\}$$

Definition 1.2 *Each Kripke structure* $\mathcal{M} = \langle W, I, J\rangle$ *gives rise to a function*

$$\mathcal{E}_{\mathcal{M}}[\![-]\!] : \textbf{\textit{Form}} \rightarrow \mathcal{P}(W),$$

where $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!]$ *is the set of worlds in which* φ *is considered true.* $\mathcal{E}_{\mathcal{M}}[\![\varphi]\!]$ *is defined inductively on the structure of* φ*, as shown in Figure 1.1.*

Note that, in the definition of $\mathcal{E}_{\mathcal{M}}[\![P \textsf{ says } \varphi]\!]$*,* $J(P)(w)$ *is simply the image of world* w *under the relation* $J(P)$*.* ▪

1.3 Inference Rules

In practice, relying on the Kripke semantics alone to reason about policies, concepts of operations (CONOPS), and behavior is inconvenient. Instead, inference rules are used to manipulate formulas in the logic. All logical rules must be sound to maintain consistency.

Definition 1.3 *A rule of form* $\dfrac{H_1 \cdots H_n}{C}$ *is sound if, for all Kripke structures* $\mathcal{M} = \langle W, I, J\rangle$*, if* $\mathcal{E}_{\mathcal{M}}[\![H_i]\!] = W$ *for each* $i \in \{1, \ldots, n\}$*, then* $\mathcal{E}_{\mathcal{M}}[\![C]\!] = W$*.* ▪

The rules in Figures 1.2 and 1.3 are all sound. As an additional check, the logic and rules have been implemented in the HOL-4 (Higher Order Logic) theorem prover as a conservative extension of the HOL logic [GM93].

$$
\begin{aligned}
\mathcal{E}_{\mathcal{M}}[\![p]\!] &= I(p)\\
\mathcal{E}_{\mathcal{M}}[\![\neg\varphi]\!] &= W - \mathcal{E}_{\mathcal{M}}[\![\varphi]\!]\\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \wedge \varphi_2]\!] &= \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!]\\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \vee \varphi_2]\!] &= \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!] \cup \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!]\\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \supset \varphi_2]\!] &= (W - \mathcal{E}_{\mathcal{M}}[\![\varphi_1]\!]) \cup \mathcal{E}_{\mathcal{M}}[\![\varphi_2]\!]\\
\mathcal{E}_{\mathcal{M}}[\![\varphi_1 \equiv \varphi_2]\!] &= \mathcal{E}_{\mathcal{M}}[\![\varphi_1 \supset \varphi_2]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\varphi_2 \supset \varphi_1]\!]\\
\mathcal{E}_{\mathcal{M}}[\![P \Rightarrow Q]\!] &= \begin{cases} W, & \text{if } J(Q) \subseteq J(P)\\ \emptyset, & \text{otherwise}\end{cases}\\
\mathcal{E}_{\mathcal{M}}[\![P \textsf{ says } \varphi]\!] &= \{w | J(P)(w) \subseteq \mathcal{E}_{\mathcal{M}}[\![\varphi]\!]\}\\
\mathcal{E}_{\mathcal{M}}[\![P \textsf{ controls } \varphi]\!] &= \mathcal{E}_{\mathcal{M}}[\![(P \textsf{ says } \varphi) \supset \varphi]\!]\\
\mathcal{E}_{\mathcal{M}}[\![P \textsf{ reps } Q \textsf{ on } \varphi]\!] &= \mathcal{E}_{\mathcal{M}}[\![P \mid Q \textsf{ says } \varphi \supset Q \textsf{ says } \varphi]\!]
\end{aligned}
$$

Figure 1.1: Semantics

1.4 Confidentiality and Integrity Policies

Confidentiality and integrity policies such as Bell-LaPadula [BL73] and Biba's Strict Integrity policy [Bib75], depend on classifying, i.e., assigning a confidentiality or integrity level to information, subjects, and objects. It is straightforward to extend the access-control logic to include confidentiality, integrity, or availability levels as needed. In what follows, we show how the syntax and semantics of *integrity* levels are added to the core access-control logic. The same process is used for levels used for *confidentiality* and *availability*.

Syntax The first step is to introduce syntax for describing and comparing security levels. **IntLabel** is the collection of *simple integrity labels*, which are used as names for the integrity levels (e.g., HI and LO).

Often, we refer abstractly to a principal P's integrity level. We define the larger set **IntLevel** of *all* possible integrity-level expressions:

$$\textbf{IntLevel} \quad ::= \quad \textbf{IntLabel} \;/\; \textsf{ilev}(\textbf{PName}).$$

An integrity-level expression is either a simple integrity label or an expression of the form $\textsf{ilev}(A)$, where A is a *simple principal name*. Informally, $\textsf{ilev}(A)$ refers to the integrity level of a simple principal A. Note, we do not define (and leave unspecified) the definition of mapping of *compound* principal expressions to labels.

Finally, we extend our definition of well-formed formulas to support comparisons of integrity levels:

$$\textbf{Form} \quad ::= \quad \textbf{IntLevel} \leq_i \textbf{IntLevel} \;/\; \textbf{IntLevel} =_i \textbf{IntLevel}$$

$$\textit{Taut} \quad \frac{}{\varphi} \quad \text{if } \varphi \text{ is an instance of a prop-logic tautology}$$

$$\textit{Modus Ponens} \quad \frac{\varphi \quad \varphi \supset \varphi'}{\varphi'} \qquad \textit{Says} \quad \frac{\varphi}{P \textsf{ says } \varphi}$$

$$\textit{MP Says} \quad \frac{}{(P \textsf{ says } (\varphi \supset \varphi')) \supset (P \textsf{ says } \varphi \supset P \textsf{ says } \varphi')}$$

$$\textit{Speaks For} \quad \frac{}{P \Rightarrow Q \supset (P \textsf{ says } \varphi \supset Q \textsf{ says } \varphi)}$$

$$\textit{Quoting} \quad \frac{}{P \mid Q \textsf{ says } \varphi \equiv P \textsf{ says } Q \textsf{ says } \varphi}$$

$$\textit{\&Says} \quad \frac{}{P\&Q \textsf{ says } \varphi \equiv P \textsf{ says } \varphi \wedge Q \textsf{ says } \varphi}$$

$$\textit{Idempotency of} \Rightarrow \quad \frac{}{P \Rightarrow P} \qquad \textit{Monotonicity of} \mid \quad \frac{P' \Rightarrow P \quad Q' \Rightarrow Q}{P' \mid Q' \Rightarrow P \mid Q}$$

$$\textit{Associativity of} \mid \quad \frac{P \mid (Q \mid R) \textsf{ says } \varphi}{(P \mid Q) \mid R \textsf{ says } \varphi}$$

$$P \textsf{ controls } \varphi \quad \stackrel{\text{def}}{=} \quad (P \textsf{ says } \varphi) \supset \varphi$$

$$P \ \textsf{ reps } Q \textsf{ on } \varphi \stackrel{\text{def}}{=} P \mid Q \textsf{ says } \varphi \supset Q \textsf{ says } \varphi$$

Figure 1.2: Core Inference Rules

Informally, a formula such as $\textsc{lo} \leq_i \textsf{ilev}(\mathit{Kate})$ states that Kate's integrity level is greater than or equal to (i.e., *dominates*) the integrity level $\textsc{lo}$. Similarly, a formula such as $\textsf{ilev}(\mathit{Barry}) =_i \textsf{ilev}(\mathit{Joe})$ states that Barry and Joe have the same integrity level.

Semantics Providing formal and precise meanings for the newly added syntax requires us to extend our Kripke structures with additional components that describe integrity classification levels. Specifically, we introduce extended Kripke structures of the form

$$\mathcal{M} = \langle W, I, J, K, L, \preceq \rangle,$$

where:

- W, I, and J are as defined earlier.
- K is a non-empty set, which serves as the universe of *integrity levels*.
- $L : (\mathbf{IntLabel} \cup \mathbf{PName}) \to K$ is a function that maps each integrity label and each simple principal name to a integrity level. L is extended to work over arbitrary integrity-level expressions, as follows:

  $$L(\textsf{ilev}(A)) = L(A),$$

  for every simple principal name A.

$$\textit{Quoting (1)} \quad \frac{P \mid Q \textsf{ says } \varphi}{P \textsf{ says } Q \textsf{ says } \varphi} \qquad \textit{Quoting (2)} \quad \frac{P \textsf{ says } Q \textsf{ says } \varphi}{P \mid Q \textsf{ says } \varphi}$$

$$\textit{Controls} \quad \frac{P \textsf{ controls } \varphi \quad P \textsf{ says } \varphi}{\varphi} \qquad \textit{Derived Speaks For} \quad \frac{P \Rightarrow Q \quad P \textsf{ says } \varphi}{Q \textsf{ says } \varphi}$$

$$\textit{Reps} \quad \frac{Q \textsf{ controls } \varphi \quad P \textsf{ reps } Q \textsf{ on } \varphi \quad P \mid Q \textsf{ says } \varphi}{\varphi}$$

$$\textit{Rep Says} \quad \frac{P \textsf{ reps } Q \textsf{ on } \varphi \quad P \mid Q \textsf{ says } \varphi}{Q \textsf{ says } \varphi}$$

Figure 1.3: Derived Rules Used in this Report

$$\ell_1 =_i \ell_2 \stackrel{\text{def}}{=} (\ell_1 \leq_i \ell_2) \wedge (\ell_2 \leq_i \ell_1)$$

$$\textit{Reflexivity of } \leq_i \quad \frac{}{\ell \leq_i \ell}$$

$$\textit{Transitivity of } \leq_i \quad \frac{\ell_1 \leq_i \ell_2 \qquad \ell_2 \leq_i \ell_3}{\ell_1 \leq_i \ell_3}$$

$$sl \leq_i \quad \frac{\textsf{ilev}(P) =_i \ell_1 \quad \textsf{ilev}(Q) =_i \ell_i \quad \ell_1 \leq_i \ell_2}{\textsf{ilev}(P) \leq_i \textsf{ilev}(Q)}$$

Figure 1.4: Inference rules for relating integrity levels

- $\preceq \subseteq K \times K$ is a partial order on K: that is, $\preceq$ is *reflexive* (for all $k \in K$, $k \preceq k$), *transitive* (for all $k_1, k_2, k_3 \in K$, if $k_1 \preceq k_2$ and $k_2 \preceq k_3$, then $k_1 \preceq k_3$), and *anti-symmetric* (for all $k_1, k_2 \in K$, if $k_1 \preceq k_2$ and $k_2 \preceq k_1$, then $k_1 = k_2$).

Using these extended Kripke structures, we extend the semantics for our new well-formed expressions as follows:

$$\begin{aligned} \mathcal{E}_{\mathcal{M}}[\![\ell_1 \leq_i \ell_2]\!] &= \begin{cases} W, & \text{if } L(\ell_1) \preceq L(\ell_2) \\ \emptyset, & \text{otherwise} \end{cases} \\ \mathcal{E}_{\mathcal{M}}[\![\ell_1 =_i \ell_2]\!] &= \mathcal{E}_{\mathcal{M}}[\![\ell_1 \leq_i \ell_2]\!] \cap \mathcal{E}_{\mathcal{M}}[\![\ell_2 \leq_i \ell_1]\!]. \end{aligned}$$

As these definitions suggest, the expression $\ell_1 =_i \ell_2$ is simply an abbreviation for $(\ell_1 \leq_i \ell_2) \wedge (\ell_2 \leq_i \ell_1)$.

Logical Rules Based on the extended Kripke semantics we introduce logical rules that support the use of integrity levels to reason about access requests. Specifically, the definition, reflexivity, and transitivity rules in Figure 1.4 reflect that $\leq_i$ is a partial order. The fourth rule is derived and convenient to have.

1.	P controls φ	assumption
2.	P says φ	assumption
3.	$(P$ says $\varphi) \supset \varphi$	1 Definition of Controls
4.	φ	2, 3 Modus Ponens

Figure 1.5: Proof of Controls Inference Rule

1.5 Examples

The following two examples in Sections 1.5.1 and 1.5.2 illustrate the use of the core inference rules to derive new inference rules. The first example proves the *Controls* rule. The second example is a more involved example illustrating the use of partial orders on integrity labels to make access-control decisions.

1.5.1 A simple example

We prove the *Controls* rule in Figure 1.3 using the core inference rules in Figure 1.2. The proof is simple and is shown in Figure 1.5. The first two lines are the assumptions of the *Controls* rule. The third and fourth lines are derived as applications the core inference rules *Definition of Controls* and *Modus Ponens*.

1.5.2 An Integrity Example

This a simple example of Biba's Strict Integrity model, [Bib75]. In this model a subject S can only read or take from an object O if the object's integrity level is at least as high or greater than S's. Similarly, subject S can only write or modify O if S's integrity level is at least as high or greater than O's. This policy prevents contamination or corruption of subjects and objects from the standpoint of quality or integrity.

Consider the following scenario. There are two grades of gas: P for premium gas and R for regular gas. There are two pumps: *Pump1* and *Pump2*. There are two cars: a car that uses regular gas (*RGC*) and a car that requires premium gas (*PGC*).

We assume the typical relation on gas grades: premium gas is higher quality than regular gas, i.e., $R \leq_i P$. We also assume cars specified as taking a particular grade of gas can safely take that grade of gas or higher. In our example, *RGC* can be fueled with regular (R) gas or premium (P) gas, whereas *PGC* can only be fueled with premium (P) gas.

Figure 1.6 diagrams the subjects, objects, and types of access permitted in this scenario. Informally, subjects *act* on objects. The subjects in the scenario are *Pump1* and *Pump2*. The objects are the Premium Gas Tank *(PGT)*, the Regular Gas Tank *(RGT)*, the Premium Gas Car *(PGC)*, and the Regular Gas Car *(RGC)*. What the diagram shows is that *Pump1* can only take or pump premium gas. It can put gas into both the regular and premium gas cars. *Pump2* can take or draw from both premium and regular gas tanks, but it can only put gas into the regular

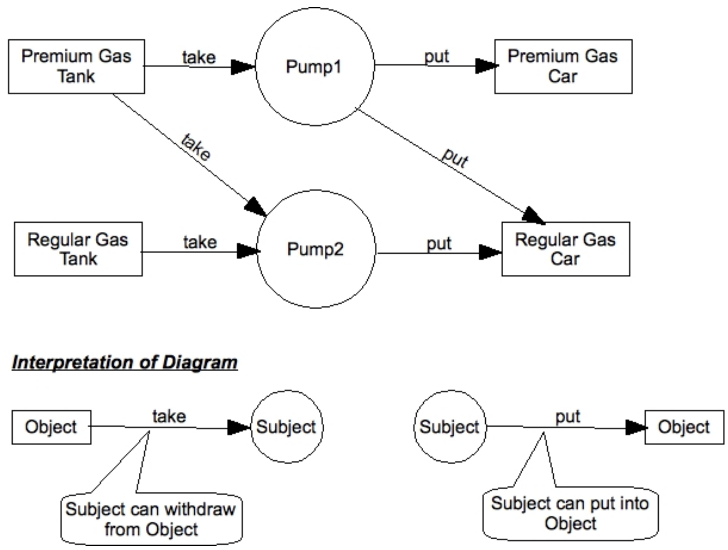

Figure 1.6: Access Diagram for Gas

Subjects and Objects	*Integrity Level*
Premium Gas Tank *PGT*	P
Regular Gas Tank *RGT*	R
Pump1	P
Pump2	R
Premium Gas Car *PGC*	P
Regular Gas Car *RGC*	R

Table 1.1: Integrity Level Assignments

gas car.

If we use P as the integrity level for premium grade and R as the integrity level for regular grade, the subjects and objects have the integrity level assignments shown in Table 1.1. These assignments combined with the access privileges shown in Figure 1.6 result in the access-control matrix in Table 1.2.

Under the assumption that the integrity levels are *partially ordered*, i.e., $R \leq_i R$, $P \leq_i P$, and $R \leq_i P$, we see that the assignment of access rights in Table 1.2 conforms with Biba's Strict Integrity policy. No car, in this assignment of *take* and *put* privileges, can be filled with gas of lesser quality than it is specified to take.

Using the access-control logic, we can formally represent the access-control policy as shown in Figure 1.7, where each formula describes the conditions under which it is permitted for Pump1 or Pump2 to exercise a take or put privilege on an object.

As an illustration, consider a request by *Pump1* to put gas into the premium-gas car *PGC*. This request is formally represented as

$$Pump1 \text{ says } \langle put, PGC \rangle.$$

Subject	PGT (P)	RGT (R)	PGC (P)	RGC (R)
Pump1 (P)	take	-	put	put
Pump2 (R)	take	take	-	put

Table 1.2: Access-Control Matrix

$$\begin{array}{l}
\mathsf{ilev}(Pump1) \leq_i \mathsf{ilev}(PGT) \supset Pump1 \text{ controls } \langle take, PGT\rangle \\
\mathsf{ilev}(PGC) \leq_i \mathsf{ilev}(Pump1) \supset Pump1 \text{ controls } \langle put, PGC\rangle \\
\mathsf{ilev}(RGC) \leq_i \mathsf{ilev}(Pump1) \supset Pump1 \text{ controls } \langle put, RGC\rangle \\
\mathsf{ilev}(Pump2) \leq_i \mathsf{ilev}(PGT) \supset Pump2 \text{ controls } \langle take, PGT\rangle \\
\mathsf{ilev}(Pump2) \leq_i \mathsf{ilev}(RGT) \supset Pump2 \text{ controls } \langle take, RGT\rangle \\
\mathsf{ilev}(RGC) \leq_i \mathsf{ilev}(Pump2) \supset Pump2 \text{ controls } \langle put, RGC\rangle
\end{array}$$

Figure 1.7: Access-Control Policy for Pumping Gas

We wish to determine if we should honor the request, i.e., if we can conclude $\langle put, PGC\rangle$. The relevant access-policy statement from Table 1.2 is

$$\mathsf{ilev}(PGC) \leq_i \mathsf{ilev}(Pump1) \supset Pump1 \text{ controls } \langle put, PGC\rangle.$$

From Table 1.1 we get the integrity level assignments of *Pump1* and *PGC*:

$$\mathsf{ilev}(PGC) =_i P \text{ and } \mathsf{ilev}(Pump1) =_i P.$$

The request with the policy and level assignment statements is enough to prove a theorem or derived inference rule justifying letting *Pump1* put gas into *PGC*. The derived inference rule is below. Its proof is shown in Figure 1.8.

$$\frac{\begin{array}{c} Pump1 \text{ says } \langle put, PGC\rangle \\ \mathsf{ilev}(PGC) \leq_i \mathsf{ilev}(Pump1) \supset Pump1 \text{ controls } \langle put, PGC\rangle \\ \mathsf{ilev}(PGC) =_i P \qquad \mathsf{ilev}(Pump1) =_i P \end{array}}{\langle put, PGC\rangle}.$$

1.	$Pump1$ says $\langle put, PGC\rangle$	request
2.	$\mathsf{ilev}(PGC) \leq_i \mathsf{ilev}(Pump1) \supset Pump1$ controls $\langle put, PGC\rangle$	integrity policy
3.	$\mathsf{ilev}(PGC) =_i P$	level assignment
4.	$\mathsf{ilev}(Pump1) =_i P$	level assignment
5.	$\mathsf{ilev}(P) \leq_i \mathsf{ilev}(P)$	Reflexivity of $\leq_i$
6.	$\mathsf{ilev}(PGC) \leq_i \mathsf{ilev}(Pump1)$	3, 4, 5 $\leq_i$ Subst
7.	$Pump1$ controls $\langle put, PGC\rangle$	2, 6 Modus Ponens
8.	$\langle put, PGC\rangle$	7, 1 Controls

Figure 1.8: Pump1 Proof

Chapter 2

Access-Control Logic in HOL

The previous sections introduced the application of an access-control logic and structural operational semantics to formally describe and reason about security and behavior. While the proofs we presented were straightforward and easily comprehended, the practical question of *how to assure and certify the proofs are correct* is important to answer. Almost every design has enough details that make pencil and paper proofs cumbersome to manage. People are human and even the best, brightest, and most experienced people can and do make mistakes.

To address these concerns, we use the Higher-Order Logic (HOL) proof checker, [GM93]. HOL is used by verification engineers in much the same way as spreadsheet programs are used by accountants. Logical definitions are entered into HOL. These definitions extend existing logical theories in HOL. Based on these definitions and existing theories containing their own definitions and theorems, new theorems are postulated as goals and verifiers attempt to prove the goals to be true. If proved true, then the goals are considered to be theorems.

What follows is not intended to be a tutorial on how to use HOL, rather it is intended to show examples of how HOL is used to check our results. HOL is extensively documented: HOL Tutorial [HOLd], HOL Description [HOLa], HOL Logic [HOLb], and HOL Reference [HOLc]. The source code for building the HOL system is freely available at http://hol.sourceforge.net/.

Everything presented here is fully described in the appendices. All the source code necessary for constructing the HOL theories and inference rules is in Appendix D. Pretty-printed listings of each HOL theory in terms of its datatypes, definitions, and theorems are in Appendix C. Descriptions of the HOL implementation of 36 access-control logic inference rules are in Appendix A. These descriptions include the name, source file, type signature, synopsis, description, failure conditions, application example, and implementation of each inference rule.

2.1 Implementation of the Access-Control Logic in HOL

In this section we give a brief overview of the access-control logic in HOL. We include as an illustration the gas station example discussed earlier.

Defining the Syntax of the Access-Control Logic in HOL We use `Hol_datatype` to introduce the syntax of the access-control logic into the HOL system. The first three types of expressions we introduce are *principal* expressions (`Princ`), *integrity level* expressions

(`IntLevel`), and *security level* expressions (`SecLevel`). The actual HOL code is shown below.

HOL supports *polymorphism*, i.e., the ability of expressions to have the same form but accept values of different type. For example, *lists* have the same structure. Either they are empty, or if they are non-empty, then they all have a head element followed by the rest of the list. Lists are polymorphic in the sense that lists of numbers, strings, bank accounts, tokens, etc., all have the same structure, but can have different types of elements.

Polymorphism in HOL is accomplished by the use of *type variables*. In HOL, type variables all start with a single quote `'`. For example, the polymorphic variable `x:'a` in HOL is a variable x whose type is given by type variable `'a`. When a specific type, e.g., `num` or `bool`, is instantiated into `'a`, then all terms of type `'a` will be instantiated with the same type.

Recall that there are three types of principal expressions:

$$P ::= A \;/\; P\&Q \;/\; P \mid Q,$$

where A is the set of simple principal names, P and Q are principal expressions, $P\&Q$ represents two principals together, and $P \mid Q$ is the compound principal P quoting Q.

In HOL, we use the type constructor `Name` to construct elements of type `Princ` from elements of type `'apn`, where `'apn` is a type variable that is instantiated to specific types such as numbers, strings, lists, etc.

For example, if we wish to construct simple principal names out of numbers we could write `Name 123456`, whose type would be `num Princ`. We could use strings as principal names. For example, `Name "Alice"` has type `string Princ`. The remaining operators in principal expressions closely correspond to what is in [CO11]. `meet` and `quoting` in HOL corresponds to & and | in [CO11].

In a similar fashion to principal names, we take advantage of polymorphism and type variables for building integrity and security levels, `IntLevel` and `SecLevel`. For example, suppose we have an integrity classification with two levels, *Prem* for premium and *Reg* for regular. Suppose also that we have previously defined these classification levels as the type *IClass*, i.e.,

$$IClass ::= Prem \;/\; Reg.$$

As there are many possible classification systems, we parameterized both integrity and security levels in the access-control logic using polymorphic type variables, in the case `'il` and `'sl` for integrity and security levels, respectively. In the case of integrity levels, we can have `iLab Prem` and `iLab Reg` as integrity labels.

As simple principals are assigned integrity or security levels, we need a way to map simple principals to integrity and security levels. This is done with the functions `il` and `sl`, respectively. For example, consider the case where simple principals are constructed from strings, e.g., `Name "Alice"`. We refer to Alice's integrity and security levels by `il (Name "Alice")` and `sl (Name "Alice")`, respectively.

The above examples show that the types `IntLevel` and `SecLevel` are each parameterized by two type variables:

1. the underlying type of simple principal names: `'apn`, (where `'apn` is thought of as an abbreviation for "a principal name"), and

2. `'il` (or `'sl`) for the particular integrity (security) classification levels used, (where `'il` and `'sl` are thought of as abbreviations for "integrity level" and "security level"), respectively.

The following HOL session shows the definition in HOL of principal expressions, integrity levels, and security levels.

```
val _ = Hol_datatype                                    1
    `Princ = Name of 'apn
           | meet of Princ => Princ
           | quoting of Princ => Princ;

     IntLevel = iLab of 'il
              | il of 'apn;

     SecLevel = sLab of 'sl
              | sl of 'apn`;
```

We can now define the abstract syntax of formulas (`Form`) in the access-control logic. The components of access-control logic formulas in HOL closely corresponds to the syntax introduced in [CO11] with a few differences due to how new types are introduced in HOL.

1. `TT` and `FF` represent *true* and *false*

2. `prop 'aavar` represents *primitive propositions* parameterized by type variable `:'aavar` (note: we chose the name `'aavar` to ensure it appears first in the alphabetized list of type variables that parameterized formulas `Form`). For example, primitive propositions could be constructed from natural numbers (`:num`)—`prop 123`; or propositions could be constructed from strings (`:string`)—`prop "read file"`.

3. Logical negation, conjunction, disjunction, implication, and equivalence are represented by `notf, andf, orf, impf`, and `eqf`.

4. The access-control logic operators says , $\Rightarrow$, controls , and P reps Q on φ are represented by `says, speaks_for, controls`, and `reps` in HOL.

5. For comparing and assigning integrity and security levels we use the following: $il_1 \leq_i il_2$ is given by `il2 domi il1` (note the change in operand order), and $il_1 =_i il_2$ is given by `il1 eqi il2` in HOL. We pronounce `il2 domi il1` as "*il_2 dominates il_1*." We have similar syntax for security levels.

6. Finally, we have equality, partial, and total order relations on natural numbers: $n_1 = n_2, n_1 \leq n_2$ and $n_1 < n_2$. These are represented in HOL by `eqn`, `lte`, and `lt`.

The function `Hol_datatype` is used to define the syntax of access-control logic formulas as a new type in HOL, as shown below.

```
val _ = Hol_datatype                                                   2
    `Form = TT
          | FF
          | prop of 'aavar
          | notf of Form
          | andf of Form => Form
          | orf of Form => Form
          | impf of Form => Form
          | eqf of Form => Form
          | says of 'apn Princ => Form
          | speaks_for of 'apn Princ => 'apn Princ
          | controls of 'apn Princ => Form
  | reps of 'apn Princ => 'apn Princ => Form
  | domi of ('apn, 'il) IntLevel => ('apn, 'il) IntLevel
  | eqi of ('apn, 'il) IntLevel => ('apn, 'il) IntLevel
  | doms of ('apn, 'sl) SecLevel => ('apn, 'sl) SecLevel
  | eqs of ('apn, 'sl) SecLevel => ('apn, 'sl) SecLevel
  | eqn of num => num
  | lte of num => num
  | lt of num => num`;
```

The definitions of `Princ` and `Form` define *prefix* operators. These operators are converted to their *infix* form (with precedence information) by the following series of commands. Note that the number following `Infixr` sets the precedence of the operator. Higher numbers indicate greater precedence.

```
(* Change "meet" and "quoting" to infix operators *)                    3

val _ = set_fixity "meet" (Infixr 630);
val _ = set_fixity "quoting" (Infixr 620);

(* and the rest *)

val _ = set_fixity "andf" (Infixr 580);
val _ = set_fixity "orf" (Infixr 570);
val _ = set_fixity "impf" (Infixr 560);
val _ = set_fixity "eqf" (Infixr 550);
val _ = set_fixity "says" (Infixr 590);
val _ = set_fixity "speaks_for" (Infixr 615);
val _ = set_fixity "controls" (Infixr 590);
val _ = set_fixity "domi" (Infixr 590);
val _ = set_fixity "eqi" (Infixr 590);
val _ = set_fixity "doms" (Infixr 590);
val _ = set_fixity "eqs" (Infixr 590);
val _ = set_fixity "eqn" (Infixr 590);
val _ = set_fixity "lte" (Infixr 590);
val _ = set_fixity "lt" (Infixr 590);
```

Defining the Semantics of the Access-Control Logic in HOL In [CO11], the access-control logic semantics is defined using Kripke structures. The core Kripke structure consists of a non-empty set of worlds W, an interpretation function I mapping each primitive proposition to a set of worlds where the proposition is true, and a function J mapping principal expressions to a relation on worlds. In [CO11], this core Kripke structure is a three-tuple $\langle W, I, J\rangle$.

As many applications use either integrity levels, security levels, or both, the core Kripke structure is extended by two functions *imap* and *smap*, which map simple principal names to integrity and security levels, respectively. The extended Kripke structure in [CO11] is the five-tuple $\langle W, I, J, imap, smap\rangle$.

There are many possible Kripke structures, not the least of which differ in the sets of worlds W, interpretation functions I, and mapping functions J, *imap*, and *smap*. In HOL we introduce a new type called `Kripke`, which is parameterized by the following type variables:

- a non-empty set of worlds of arbitrary type, in HOL this is the type variable `'aaworld`,
- a non-empty set of primitive propositions of arbitrary type `'aavar`,
- a non-empty set of simple principal names `'apn`,
- a set of integrity levels of arbitrary type `'il`, and
- a set of security levels of arbitrary type `'sl`.

In HOL, the type `Kripke` is created by type constructor `KS` applied to three functions in the following order:

1. the interpretation function I, whose type signature is `'aavar -> 'aaworld set`,

2. the mapping J from principals to a relation on worlds, whose type signature is `'apn -> ('aaworld -> ('aaworld set))`,

3. the mapping function *imap* from simple principal names to integrity levels, whose type signature is `'apn -> 'il`, and

4. the mapping function *smap* from simple principal names to security levels, whose type signature is `'apn -> 'sl`.

In HOL, the Kripke structures are built by expressions of the form: `KS I J imap smap`. The universe of worlds W is omitted because W is included in the type signatures of I and J.

The actual HOL code that defines type `Kripke` in HOL is below.

```
val _ = Hol_datatype                                                  4
    `Kripke = KS of ('aavar -> ('aaworld set)) =>
                  ('apn -> ('aaworld -> ('aaworld set))) =>
    ('apn -> 'il) => ('apn -> 'sl)`;
```

Once type `Kripke` is defined, we define accessor functions to retrieve the various components of Kripke structures.

$$\begin{aligned} intpKS(KS\ Intp\ Jfn\ ilmap\ slmap) &= Intp \\ jKS(KS\ Intp\ Jfn\ ilmap\ slmap) &= Jfn \\ imapKS(KS\ Intp\ Jfn\ ilmap\ slmap) &= ilmap \\ smapKS(KS\ Intp\ Jfn\ ilmap\ slmap) &= slmap. \end{aligned}$$

The definitions of the accessor functions in HOL are as follows.

```
val intpKS_def =                                                      5
    Define `intpKS(KS Intp Jfn ilmap slmap) = Intp`;

val jKS_def =
    Define `jKS(KS Intp Jfn ilmap slmap) = Jfn`;

val imapKS_def =
    Define `imapKS(KS Intp Jfn ilmap slmap) = ilmap`;

val smapKS_def =
    Define `smapKS(KS Intp Jfn ilmap slmap) = slmap`;
```

With the above definitions it is straightforward to prove for any Kripke structure M deconstructed using the accessor functions, M can be reassembled using `KS`, i.e.,

$\vdash \forall M.\ M$ = KS (intpKS M) (jKS M) (imapKS M) (smapKS M).

With the definitions of Kripke structures and the syntax of principal expressions and formulas defined in HOL, we can define the semantics of formulas in HOL corresponding to Figure 1.1.

Extended mapping from principal expressions to relations on worlds The mapping function `J` to which `KS` is applied maps *simple principal names* to relations on worlds `'aaworld -> ('aaworld set)`. Note: the type of `J` differs slightly from the definition of J in [CO11]. In particular, the type of J in [CO11] is a function from principal names to the set $\mathcal{P}(W \times W)$. Here, the type of `J` is `PName -> 'aaworld -> ('aaworld set)`. This type is a bit more convenient than the type of J to fetch the set of worlds to which any particular world is related.

Recall, the *extended* mapping function $\hat{J}$ extends J to compound principal as shown below.

$$\begin{aligned} \hat{J}(A) &= J, \text{ where A is a simple principal,} \\ \hat{J}(P\&Q) &= \hat{J}(P) \cup \hat{J}(Q), \text{ and} \\ \hat{J}(P \mid Q) &= \hat{J}(P) \circ \hat{J}(Q). \end{aligned}$$

The definition in HOL is shown below.

```
val Jext_def =                                                      6
    Define
    `(Jext (J:'pn -> 'w ->'w set) (Name s) = J s) /\
     (Jext J (P1 meet P2) = ((Jext J P1) RUNION (Jext J P2))) /\
     (Jext J (P1 quoting P2) = (Jext J P2) O (Jext J P1))`;
```

Defining Integrity and Security Levels and Partial Orders Introducing integrity and security levels into the HOL implementation of the access-control logic comes next. We define two functions `Lifn` and `Lsfn` that map integrity and security labels to levels, and map simple principal names to levels. In the case of integrity and security labels `iLab l` and `sLab l`, the level returned is just `l`. In the case of simple names, the mapping of names to levels is specified by the `ilmap` and `slmap` functions that are part of the Kripke structure M. Specifically,

$$\begin{aligned} ilmap &= imapKS\ M \\ slmap &= smapKS\ M. \end{aligned}$$

The definition of `Lifn` and `Lsfn` are as follows:

$\vdash (\forall M\ l.$ Lifn M (iLab l) = l) $\wedge\ \forall M\ name.$ Lifn M (il $name$) = imapKS $M\ name$

$\vdash (\forall M\ l.$ Lsfn M (sLab l) = l) $\wedge\ \forall M\ name.$ Lsfn M (sl $name$) = smapKS $M\ name$.

Their corresponding definitions in HOL are below.

```
val Lifn_def =                                              7
    Define
    `(Lifn M (iLab l) = l) /\
     (Lifn M (il name) = imapKS M name)`;
```

```
val Lsfn_def =                                              8
    Define
    `(Lsfn M (sLab l) = l) /\
     (Lsfn M (sl name) = smapKS M name)`;
```

The Bell-LaPadula security model and Biba's Strict Integrity model use a *partial ordering* of integrity or security levels. In HOL, partial orders are defined by `WeakOrder`. WeakOrder and its components are defined below.

$\vdash \forall Z.\ \mathrm{WeakOrder}\ Z \iff \mathrm{reflexive}\ Z \land \mathrm{antisymmetric}\ Z \land \mathrm{transitive}\ Z$, where
$\vdash \forall R.\ \mathrm{reflexive}\ R \iff \forall x.\ R\ x\ x$,
$\vdash \forall R.\ \mathrm{antisymmetric}\ R \iff \forall x\ y.\ R\ x\ y \land R\ y\ x \Rightarrow (x = y)$, and
$\vdash \forall R.\ \mathrm{transitive}\ R \iff \forall x\ y\ z.\ R\ x\ y \land R\ y\ z \Rightarrow R\ x\ z$.

We introduce a new type `po` (partial order) in HOL. The idea is to have `po` be polymorphic like other types such as *lists*, e.g., `num list`—lists of natural numbers. If r is a partially ordered relation defined on a set of classification labels defined in HOL as the type `class`, then we can introduce the HOL type `class po`. To introduce the type `'a po` in HOL is a three-step process.

1. The HOL predicate `WeakOrder` is used to select partial orderings from relations on type `'a`.

2. We prove a theorem stating that the new type `'a po` has at least one member.

3. We introduce the new type using `new_type_definition` in HOL using the theorem that states `'a po` is non-empty.

The first step in HOL is to prove that there exists a relation on `'a` that satisfies `WeakOrder`. The HOL proof shown below proves a theorem named EQ_WeakOrder, which states equality ($= is the prefix form) is a partial order. The theorem and proof code in HOL follow.

$\vdash$ WeakOrder $=

```
val EQ_WeakOrder =                                          9
    store_thm("EQ_WeakOrder",
Term `WeakOrder ($=)`,
REWRITE_TAC
(map (SPEC ``($=):('a->'a->bool)``)
[(INST_TYPE [Type`:'g` |-> Type `:'a`] WeakOrder),
 reflexive_def, antisymmetric_def,transitive_def]) THEN
   PROVE_TAC []);
```

Using the EQ_WeakOrder theorem we can prove easily that there exists at least one relation satisfying WeakOrder. The theorem and proof code follow.

$\vdash \exists R.$ WeakOrder R

```
val WeakOrder_Exists =                                              10
    save_thm
    ("WeakOrder_Exists",
     (EXISTS (Term `?R.WeakOrder R`, Term `$=`) EQ_WeakOrder));
```

The theorem WeakOrder_Exists is enough for HOL to conclude that the type `:'a po` is non-empty. We introduce `po` as a new type using `new_type_definition` as shown below.

```
val po_type_definition =                                            11
    new_type_definition ("po",WeakOrder_Exists);
```

Executing the `new_type_definition` command introduces the following type definition:

$\vdash \exists rep.$ TYPE_DEFINITION WeakOrder rep, where

$\vdash$ TYPE_DEFINITION =

$(\lambda P\ rep.\ (\forall x'\ x''.\ (rep\ x' = rep\ x'') \Rightarrow (x' = x'')) \land \forall x.\ P\ x \iff \exists x'.\ x = rep\ x')$

The above type definition establishes the existence of the type `'a po` constructed from relations of type `'a -> 'a -> bool`. Because the type `'a po` is non-empty, there is a one-to-one mapping from elements of type `'a -> 'a -> bool` to elements of type `'a po`. The HOL built-in function `define_new_type_bijections` produces the appropriate theorems given the desired names of the abstraction and representation functions (`PO` and `repPO`), and the name of the newly defined type, here it is `po_type_definition`.

```
val po_bij = save_thm ("po_bij",                                    12
     (define_new_type_bijections
      {name="po_tybij", ABS="PO", REP="repPO",
       tyax=po_type_definition}));
```

The resulting theorem po_bij, is shown below.

$\vdash (\forall (a$:'a po). PO (repPO a) = $a) \land$

$\forall (r$:'a $\to$ 'a $\to$ bool). WeakOrder $r \iff$ (repPO (PO r) = r) [po_bij]

The theorem po_bij has two statements. (1) All elements a : 'apo are mapped to their underlying representations by `repPO a` and back again by `PO`, and (2) for all partial orders `r:'a -> 'a -> bool`, `PO` composed with `repPO` is the identity function. Later on we will use both `PO` and `repPO` to define partial orders using specific integrity and security labels.

At this point, having defined Kripke structures, integrity and security levels, and partial orders, we have all the necessary components to define the semantics of access-control logic formulas. The HOL session below shows the actual definition. Note that `Oi:'il po` and

`Os:'is po` are partial orderings on integrity and security levels. `UNIV:('w)set` corresponds to W, and is non-empty as every type in HOL is non-empty. Kripke structure `M:('w,'v,'pn,'il,'is) Kripke` is parameterized by type variables `'w` for worlds, `'v` for propositional variables, `'pn` for simple principal names, `'il` for integrity levels, and `'sl` for security levels.

13

```
val Efn_def =
    Define
    `(Efn (Oi:'il po) (Os:'is po) (M:('w,'v,'pn,'il,'is) Kripke)
                TT = UNIV) /\
     (Efn Oi Os M FF = {}) /\
     (Efn Oi Os M (prop p) = ((intpKS M) p)) /\
     (Efn Oi Os M (notf f) = (UNIV DIFF (Efn Oi Os M f))) /\
     (Efn Oi Os M (f1 andf f2) =
           ((Efn Oi Os M f1) INTER (Efn Oi Os M f2))) /\
     (Efn Oi Os M (f1 orf f2) =
           ((Efn Oi Os M f1) UNION (Efn Oi Os M f2))) /\
     (Efn Oi Os M (f1 impf f2) =
           ((UNIV DIFF (Efn Oi Os M f1)) UNION (Efn Oi Os M f2)))  /\
     (Efn Oi Os M (f1 eqf f2) =
           ((UNIV DIFF (Efn Oi Os M f1) UNION (Efn Oi Os M f2)) INTER
            (UNIV DIFF (Efn Oi Os M f2) UNION (Efn Oi Os M  f1)))) /\
     (Efn Oi Os M(P says f) =
           {w | Jext (jKS M) P w SUBSET (Efn Oi Os M f)}) /\
     (Efn Oi Os M (P speaks_for Q) =
           (if ((Jext (jKS M) Q) RSUBSET (Jext (jKS M) P)) then UNIV else
           {})) /\
     (Efn Oi Os M(P controls f) =
           ((UNIV DIFF
           ({w | Jext (jKS M) P w SUBSET Efn Oi Os M f})) UNION
            (Efn Oi Os M f)))  /\
     (Efn Oi Os M (reps P Q f) =
           ((UNIV DIFF
           ({w | Jext (jKS M) (P quoting Q) w SUBSET
                 Efn Oi Os M f})) UNION
            {w | Jext (jKS M) Q w SUBSET Efn Oi Os M f})) /\
     (Efn Oi Os M (intl1 domi intl2) = (* note inversion 3/12/09 *)
           (if repPO Oi (Lifn M intl2) (Lifn M intl1)
           then UNIV else {})) /\
     (Efn Oi Os M (intl2 eqi intl1) = (* ** note inversion 7/30/09 ** *)
           (if repPO Oi (Lifn M intl2) (Lifn M intl1)
           then UNIV else {}) INTER
           (if repPO Oi (Lifn M intl1) (Lifn M intl2)
           then UNIV else {})) /\
     (Efn Oi Os M (secl1 doms secl2) = (* note inversion *)
           (if repPO Os (Lsfn M secl2) (Lsfn M secl1)
           then UNIV else {})) /\
     (Efn Oi Os M (secl2 eqs secl1) = (* ** note inversion ** *)
           (if repPO Os (Lsfn M secl2) (Lsfn M secl1)
           then UNIV else {}) INTER
           (if repPO Os (Lsfn M secl1) (Lsfn M secl2)
           then UNIV else {})) /\
     (Efn Oi Os M ((numExp1:num) eqn (numExp2:num)) =
           (if (numExp1 = numExp2)
            then UNIV else {})) /\
     (Efn Oi Os M ((numExp1:num) lte (numExp2:num)) =
           (if (numExp1 <= numExp2)
            then UNIV else {})) /\
     (Efn Oi Os M ((numExp1:num) lt (numExp2:num)) =
           (if (numExp1 < numExp2)
            then UNIV else {}))`;
```

After defining the semantics as shown above, individual theorems corresponding to each operator or relation in the access-control logic are proved. Figure 2.1 shows the collection of theorems that define the semantics of the access-control logic.

$\vdash \forall Oi\ Os\ M.\ \text{Efn}\ Oi\ Os\ M\ \text{TT} = \text{univ}(:\text{'v})$ [TT_def]

$\vdash \forall Oi\ Os\ M.\ \text{Efn}\ Oi\ Os\ M\ \text{FF} = \{\ \}$ [FF_def]

$\vdash \forall Oi\ Os\ M\ p.\ \text{Efn}\ Oi\ Os\ M\ (\text{prop}\ p) = \text{intpKS}\ M\ p$ [prop_def]

$\vdash \forall Oi\ Os\ M\ f.\ \text{Efn}\ Oi\ Os\ M\ (\text{notf}\ f) = \text{univ}(:\text{'v})\ \text{DIFF Efn}\ Oi\ Os\ M\ f$ [notf_def]

$\vdash \forall Oi\ Os\ M\ f_1\ f_2.\ \text{Efn}\ Oi\ Os\ M\ (f_1\ \text{andf}\ f_2) = \text{Efn}\ Oi\ Os\ M\ f_1 \cap \text{Efn}\ Oi\ Os\ M\ f_2$ [andf_def]

$\vdash \forall Oi\ Os\ M\ f_1\ f_2.\ \text{Efn}\ Oi\ Os\ M\ (f_1\ \text{orf}\ f_2) = \text{Efn}\ Oi\ Os\ M\ f_1 \cup \text{Efn}\ Oi\ Os\ M\ f_2$ [orf_def]

$\vdash \forall Oi\ Os\ M\ f_1\ f_2.\ \text{Efn}\ Oi\ Os\ M\ (f_1\ \text{impf}\ f_2) =$ [impf_def]
$\quad \text{univ}(:\text{'v})\ \text{DIFF Efn}\ Oi\ Os\ M\ f_1 \cup \text{Efn}\ Oi\ Os\ M\ f_2$

$\vdash \forall Oi\ Os\ M\ f_1\ f_2.\ \text{Efn}\ Oi\ Os\ M\ (f_1\ \text{eqf}\ f_2) =$ [eqf_def]
$\quad (\text{univ}(:\text{'v})\ \text{DIFF Efn}\ Oi\ Os\ M\ f_1 \cup \text{Efn}\ Oi\ Os\ M\ f_2) \cap$
$\quad (\text{univ}(:\text{'v})\ \text{DIFF Efn}\ Oi\ Os\ M\ f_2 \cup \text{Efn}\ Oi\ Os\ M\ f_1)$

$\vdash \forall Oi\ Os\ M\ P\ f.\ \text{Efn}\ Oi\ Os\ M\ (P\ \text{says}\ f) =$ [says_def]
$\quad \{w \mid \text{Jext}\ (\text{jKS}\ M)\ P\ w \subseteq \text{Efn}\ Oi\ Os\ M\ f\}$

$\vdash \forall Oi\ Os\ M\ P\ Q.\ \text{Efn}\ Oi\ Os\ M\ (P\ \text{speaks\_for}\ Q) =$ [speaks_for_def]
$\quad \textbf{if}\ \text{Jext}\ (\text{jKS}\ M)\ Q\ \text{RSUBSET Jext}\ (\text{jKS}\ M)\ P\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}$

$\vdash \forall Oi\ Os\ M\ P\ f.\ \text{Efn}\ Oi\ Os\ M\ (P\ \text{controls}\ f) =$ [controls_def]
$\quad \text{univ}(:\text{'v})\ \text{DIFF}\ \{w \mid \text{Jext}\ (\text{jKS}\ M)\ P\ w \subseteq \text{Efn}\ Oi\ Os\ M\ f\} \cup \text{Efn}\ Oi\ Os\ M\ f$

$\vdash \forall Oi\ Os\ M\ P\ Q\ f.\ \text{Efn}\ Oi\ Os\ M\ (\text{reps}\ P\ Q\ f) =$ [reps_def]
$\quad \text{univ}(:\text{'v})\ \text{DIFF}\ \{w \mid \text{Jext}\ (\text{jKS}\ M)\ (P\ \text{quoting}\ Q)\ w \subseteq \text{Efn}\ Oi\ Os\ M\ f\} \cup$
$\quad \{w \mid \text{Jext}\ (\text{jKS}\ M)\ Q\ w \subseteq \text{Efn}\ Oi\ Os\ M\ f\}$

$\vdash \forall Oi\ Os\ M\ intl_1\ intl_2.\ \text{Efn}\ Oi\ Os\ M\ (intl_1\ \text{domi}\ intl_2) =$ [domi_def]
$\quad \textbf{if}\ \text{repPO}\ Oi\ (\text{Lifn}\ M\ intl_2)\ (\text{Lifn}\ M\ intl_1)\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}$

$\vdash \forall Oi\ Os\ M\ intl_2\ intl_1.\ \text{Efn}\ Oi\ Os\ M\ (intl_2\ \text{eqi}\ intl_1) =$ [eqi_def]
$\quad (\textbf{if}\ \text{repPO}\ Oi\ (\text{Lifn}\ M\ intl_2)\ (\text{Lifn}\ M\ intl_1)\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}) \cap$
$\quad \textbf{if}\ \text{repPO}\ Oi\ (\text{Lifn}\ M\ intl_1)\ (\text{Lifn}\ M\ intl_2)\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}$

$\vdash \forall Oi\ Os\ M\ secl_1\ secl_2.\ \text{Efn}\ Oi\ Os\ M\ (secl_1\ \text{doms}\ secl_2) =$ [doms_def]
$\quad \textbf{if}\ \text{repPO}\ Os\ (\text{Lsfn}\ M\ secl_2)\ (\text{Lsfn}\ M\ secl_1)\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}$

$\vdash \forall Oi\ Os\ M\ secl_2\ secl_1.\ \text{Efn}\ Oi\ Os\ M\ (secl_2\ \text{eqs}\ secl_1) =$ [eqs_def]
$\quad (\textbf{if}\ \text{repPO}\ Os\ (\text{Lsfn}\ M\ secl_2)\ (\text{Lsfn}\ M\ secl_1)\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}) \cap$
$\quad \textbf{if}\ \text{repPO}\ Os\ (\text{Lsfn}\ M\ secl_1)\ (\text{Lsfn}\ M\ secl_2)\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}$

$\vdash \forall Oi\ Os\ M\ numExp_1\ numExp_2.\ \text{Efn}\ Oi\ Os\ M\ (numExp_1\ \text{eqn}\ numExp_2) =$ [eqn_def]
$\quad \textbf{if}\ numExp_1 = numExp_2\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}$

$\vdash \forall Oi\ Os\ M\ numExp_1\ numExp_2.\ \text{Efn}\ Oi\ Os\ M\ (numExp_1\ \text{lte}\ numExp_2) =$ [lte_def]
$\quad \textbf{if}\ numExp_1 \leq numExp_2\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}$

$\vdash \forall Oi\ Os\ M\ numExp_1\ numExp_2.\ \text{Efn}\ Oi\ Os\ M\ (numExp_1\ \text{lt}\ numExp_2) =$ [lt_def]
$\quad \textbf{if}\ numExp_1 < numExp_2\ \textbf{then}\ \text{univ}(:\text{'v})\ \textbf{else}\ \{\ \}$

Figure 2.1: Theorems Defining Semantics of the Access-Control Logic

2.2 Access-Control Logic Inference Rules in HOL

Our objective for embedding the access-control logic into HOL, as a conservative extension of the HOL logic, is to provide sound computer-assisted reasoning tools for access-control logic users. To do this, we provide 36 inference rules, as shown in Appendix A, which correspond closely to the inference rules in the textbook *Access Control, Security, and Trust: A Logical Approach*, [CO11].

Each inference rule is described in terms of its name, file containing its definition, synopsis, description, failure conditions, an example application, and its implementation. With few exceptions, inference rules are small functional programs that specialize and unify of the bound variables of relevant theorems.

Chapter 3

Integrity Example in HOL

We now turn to showing how the integrity example in Section 1.5 involving premium and regular gas is described and verified in HOL. Our first task is to define the integrity levels themselves, how they are related, and show that the relation is a partial order. The steps we take are the following:

1. Define the integrity levels, in this case `Reg` and `Prem`.
2. Define the relation `ICOrder` on integrity levels.
3. Prove that `ICOrder` is reflexive, antisymmetric, and transitive and thus satisfies the `WeakOrder` condition necessary for the type `(ICOrder) po`.

3.1 Introducing the Integrity Levels in HOL

We introduce the integrity levels `Prem` and `Reg` as a new datatype `IClass` in HOL.

```
val _ = Hol_datatype                                                    14
    `IClass = Prem | Reg`;
```

Next, we define the ordering on `IClass` according to Table 3.1. `Prem` dominates `Reg`, `Prem` dominates itself, and `Reg` dominates itself.

```
val ICOrder_def =                                                       15
 Define `ICOrder y x =
 if x = Prem then T else if y = Prem then F else T`;
```

With the above definition we can build what amounts to Table 3.1 in HOL as a list of theorems `ICO_table` as shown below.

y	x	ICOrder y x
Reg	Reg	T
Reg	Prem	T
Prem	Reg	F
Prem	Prem	T

Table 3.1: Ordering on Integrity Levels

```
- val ICO_table = foldl (uncurry append) []                          16
    (map (fn x => map (fn y =>
      REWRITE_CONV
        [ICOrder_def, TypeBase.distinct_of (Type`:IClass`),
                      GSYM (TypeBase.distinct_of (Type`:IClass`))]
              (Term`ICOrder ^x ^y`))
              [``Prem``,``Reg``])
     [``Prem``,``Reg``]);
> val ICO_table =
  [|- ICOrder Reg Prem <=> T, |- ICOrder Reg Reg <=> T,
   |- ICOrder Prem Prem <=> T, |- ICOrder Prem Reg <=> F] : thm list
```

From the definition of `ICOrder` it is straightforward to see that it is a partial order, i.e., antisymmetric, reflexive, and transitive.

$\vdash$ antisymmetric ICOrder [ICOrder_antisymmetric]

$\vdash$ reflexive ICOrder [ICOrder_reflexive]

$\vdash$ transitive ICOrder [ICOrder_transitive]

The following are short proofs in HOL that prove these three properties.

```
val ICOrder_antisymmetric =                                          17
store_thm
  ("ICOrder_antisymmetric",
  ``antisymmetric ICOrder``,
  REWRITE_TAC [antisymmetric_def] THEN
  Cases THEN
  Cases THEN
  REWRITE_TAC ICO_table);

val ICOrder_reflexive =
store_thm
  ("ICOrder_reflexive",
  ``reflexive ICOrder``,
  REWRITE_TAC [reflexive_def] THEN
  Cases THEN
  REWRITE_TAC ICO_table);

val ICOrder_transitive =
store_thm
  ("ICOrder_transitive",
  ``transitive ICOrder``,
  REWRITE_TAC [transitive_def] THEN
  Cases THEN
  Cases THEN
  Cases THEN REWRITE_TAC ICO_table);
```

With these three theorems, we prove that `ICOrder` satisfies `WeakOrder`, i.e., `ICOrder` is a partial order.

$\vdash$ WeakOrder ICOrder [ICOrder_WO]

The proof in HOL is shown below.

```
val ICOrder_WO =                                                     18
store_thm
   ("ICOrder_WO",
    ``WeakOrder ICOrder``,
   REWRITE_TAC [ICOrder_antisymmetric,
                ICOrder_reflexive, ICOrder_transitive,WeakOrder]);
```

With the ICOrder_WO theorem, we can define the type `ICLass_PO` as `POICOrder` and establish its properties. The HOL function and resulting definition are shown below.

```
val IClass_PO_def =                                                  19
    Define `IClass_PO = PO ICOrder`;
```

$\vdash$ IClass_PO = PO ICOrder [IClass_PO_def]

Using the definition of IClass_PO, the one-to-one properties of partial orders given by po_-bij, and the fact that ICOrder is a partial order, we can show that the representation of IClass_PO is ICOrder. The proof in HOL is shown below followed by the theorem repPO_IClass_PO.

1.	$Pump1$ says $\langle put, PGC\rangle$	assumption
2.	$\mathsf{ilev}(PGC) \leq_i \mathsf{ilev}(Pump1) \supset Pump1$ controls $\langle put, PGC\rangle$	assumption
3.	$\mathsf{ilev}(PGC) =_i P$	assumption
4.	$\mathsf{ilev}(Pump1) =_i P$	assumption
5.	$\mathsf{ilev}(P) \leq_i \mathsf{ilev}(P)$	Reflexivity of $\leq_i$
6.	$\mathsf{ilev}(PGC) \leq_i \mathsf{ilev}(Pump1)$	3, 4, 5 $\leq_i$ Subst
7.	$Pump1$ controls $\langle put, PGC\rangle$	2, 6 Modus Ponens
8.	$\langle put, PGC\rangle$	7, 1 Controls

Figure 3.1: Pump1 Proof Reprised

```
val repPO_IClass_PO =                                                20
  store_thm ("repPO_IClass_PO", Term `repPO IClass_PO = ICOrder`,
              REWRITE_TAC [IClass_PO_def, po_bij,
              EQ_MP (ISPEC (Term`ICOrder`) WO_repPO) ICOrder_WO]);
```

⊢ repPO IClass_PO = ICOrder [repPO_IClass_PO]

Finally, with respect to the partial order of integrity labels ICOrder, we combine several theorems characterizing ICOrder and name it ICOrder_simp. The HOL proof and theorem are shown below.

```
val ICOrder_simp = save_thm ("ICOrder_simp",                         21
 CONJ repPO_IClass_PO
   (CONJ ICOrder_def
      (CONJ (TypeBase.distinct_of (Type`:IClass`))
            (GSYM (TypeBase.distinct_of (Type`:IClass`))))));
```

⊢ (repPO IClass_PO = ICOrder) ∧ [ICOrder_simp]
(∀*y x*. ICOrder *y x* ⟺ **if** *x* = Prem **then** T **else if** *y* = Prem **then** F **else** T) ∧
Prem ≠ Reg ∧ Reg ≠ Prem

3.2 Gas Example in HOL

Having defined the integrity levels and the ordering ICOrder above, we now show how to formalize Figure 1.6 in HOL. The first definition we make is to define the set of operations `Put` and `Take` as a new datatype `Op`. We define an `Action` as a pair consisting of an `Op` and principal `'apn Princ`, where `'apn` is the type variable for simple principals.

```
val _ = Hol_datatype `Op = Put | Take`;                              22

val _ = Hol_datatype `Action = Act of (Op # 'apn Princ)`;
```

Given the above, subjects (principals) request to perform actions on objects (principals), which are either approved or disapproved by the reference monitors guarding the objects. We reprise the gas station proof in Figure 3.1 and show the corresponding HOL proof in the boxed session transcripts that follow. The first assumption in line 1 is the subject *Pump1* requesting to put gas into the object *PGC*, a premium gas car. In HOL, using the datatypes `Action` and `Op`, the request is introduced as theorem `a1` as follows. The `ACL_ASSUM2` inference rule takes as one of its three arguments the HOL term corresponding to the access-control logic formula $Pump_1$ says $\langle Put, PGC \rangle$. The remaining two HOL terms specify which integrity and security partial orderings to use.

23
```
val a1 =
      ACL_ASSUM2
      ``((Name "Pump1") says (prop (Act (Put,Name "PGC")))):
      (string Action, string, IClass, 'e) Form``
     ``IClass_PO`` ``Os:'e po``;
> val a1 =
     [.]
    |- (M,IClass_PO,Os) sat
       Name "Pump1" says prop (Act (Put,Name "PGC")) : thm
```

Next, line 2 of the proof in Figure 3.1 is introduced using `ACL_ASSUM2`. This is the policy on *put* access to `PGC` corresponding to Biba's strict integrity policy.

24
```
- val a2 =
      ACL_ASSUM2
      ``((il "Pump1"):(string,IClass)IntLevel domi (il "PGC")):
        (string Action, string, IClass, 'e) Form impf
        ((Name "Pump1") controls (prop (Act (Put,Name "PGC"))))``
      ``IClass_PO`` ``Os:'e po``;
> val a2 =
     [.]
    |- (M,IClass_PO,Os) sat
       il "Pump1" domi il "PGC" impf
       Name "Pump1" controls prop (Act (Put,Name "PGC")) : thm
```

The third and fourth assumptions in the proof in Figure 3.1 specify the integrity levels assigned to both `PGC` and `Pump1`. Both are assigned `P`, i.e., premium grade.

```
- val a3 =                                                          25
      ACL_ASSUM2
      ``((il "PGC") eqi
(iLab Prem)):(string Action, string, IClass, 'e) Form``
      ``IClass_PO`` ``Os:'e po``;
> val a3 =  [.] |- (M,IClass_PO,Os) sat il "PGC" eqi iLab Prem : thm
-  val a4 =
     ACL_ASSUM2
     ``((il "Pump1") eqi
        (iLab Prem)):(string Action, string, IClass, 'e) Form``
     ``IClass_PO`` ``Os:'e po``;
> val a4 =  [.] |- (M,IClass_PO,Os) sat il "Pump1" eqi iLab Prem : thm
```

From a3, a4, and the fact that ICOrder is a partial order, we can conclude that Pump1 dominates PGC. This is done by the IL_DOMI inference rule, which yields th5. This essentially combines lines 5 and 6 in Figure 3.1.

```
- val th5 = IL_DOMI ICOrder_simp a3 a4;                             26
> val th5 =  [..] |- (M,IClass_PO,Os) sat il "Pump1" domi il "PGC" : thm
```

Lines 7 and 8 in Figure 3.1 correspond to th6 and th7 below, which are derived using the ACL_MP and CONTROLS inference rules.

```
- val th6 = ACL_MP th5 a2;                                          27
> val th6 =
     [...]
    |- (M,IClass_PO,Os) sat
       Name "Pump1" controls prop (Act (Put,Name "PGC")) : thm
- val th7 = CONTROLS th6 a1;
> val th7 =  [....] |- (M,IClass_PO,Os) sat
                        prop (Act (Put,Name "PGC")) : thm
```

Finally, DISCH_ALL is used to discharge all the assumptions and move them into the antecedent of the conclusion. This is th8.

```
- val th8 = DISCH_ALL th7;                                          28
> val th8 =
    |- (M,IClass_PO,Os) sat
       Name "Pump1" says prop (Act (Put,Name "PGC")) ==>
       (M,IClass_PO,Os) sat
       il "Pump1" domi il "PGC" impf
       Name "Pump1" controls prop (Act (Put,Name "PGC")) ==>
       (M,IClass_PO,Os) sat il "Pump1" eqi iLab Prem ==>
       (M,IClass_PO,Os) sat il "PGC" eqi iLab Prem ==>
       (M,IClass_PO,Os) sat prop (Act (Put,Name "PGC")) : thm
```

Table 3.2 summarizes the access-control rules in HOL used to do the proof above. A complete listing of access-control logic rules appears in Appendix A.

Access-Control Inference Rules		
assumption	*ACL_ASSUM f*	$\dfrac{}{(M,O_i,O_s) \text{ sat } f \vdash (M,O_i,O_s) \text{ sat } f}$
assumption with specific partial orders O_i' and O_s'	*ACL_ASSUM2 f* O_i' O_s'	$\dfrac{}{(M,O_i',O_s') \text{ sat } f \vdash (M,O_i',O_s') \text{ sat } f}$
Reflexivity of $\leq_i$	*IL_DOMI*	$\dfrac{\begin{array}{c} A_1 \vdash (M,O_i,O_s) \text{ sat } \mathit{il}\ P\ \mathit{eqi}\ l_1 \\ A_2 \vdash (M,O_i,O_s) \text{ sat } \mathit{il}\ Q\ \mathit{eqi}\ l_2 \\ A_3 \vdash (M,O_i,O_s) \text{ sat } l_2\ \mathit{domi}\ l_1 \end{array}}{\{A_1 \cup A_2 \cup A\_3\} \vdash (M,O_i,O_s) \text{ sat } \mathit{il}\ Q\ \mathit{domi}\ \mathit{il}\ P}$
Modus Ponens	*ACL_MP*	$\dfrac{\begin{array}{c} A_1 \vdash (M,O_i,O_s) \text{ sat } f_1 \\ A_2 \vdash (M,O_i,O_s) \text{ sat } f_1 \text{ impf } f_2 \end{array}}{A_1 \cup A_2 \vdash (M,O_i,O_s) \text{ sat } f_2}$
Controls	*CONTROLS*	$\dfrac{\begin{array}{c} A_1 \vdash (M,O_i,O_s) \text{ sat } P \text{ controls } f \\ A_2 \vdash (M,O_i,O_s) \text{ sat } P \text{ says } f \end{array}}{A_1 \cup A_2 \vdash (M,O_i,O_s) \text{ sat } f}$

Table 3.2: Access-Control Logic Inference Rules and HOL Inference Rules

Appendices

Appendix A

Access-Control Logic Inference Rules in HOL

A.1 ACL_ASSUM

ACL_ASSUM	(acl_infRules)

ACL_ASSUM : term -> thm

Synopsis

Introduces an assumption in the access-control logic.

Description

When applied to a term f, which must have type Form, ACL_ASSUM introduces a theorem

(M,Oi,Os) sat f |- (M,Oi,Os) sat f.

```
 --------------------------------  ACL_ASSUM f
(M,Oi,Os) sat f |- (M,Oi,Os) sat f
```

Failure

Fails unless f has type Form.

Example

The following application:

```
- val a1 =
      ACL_ASSUM
      ``(Token:'c Princ) says (Role says f:('a,'c,'d,'e)Form)``;
```

produces the following result:

```
val a1 =  [.] |- (M,Oi,Os) sat Token says Role says f : thm
```

Implementation

The implementation is as follows

```
fun ACL_ASSUM f =
let
 val f_type = type_of f
 val f_type_parts = dest_type f_type
 val [prop_type, name_type, integ_type, sec_type] = snd f_type_parts
 val M_type =
   mk_type ("Kripke",
            [prop_type, ``:'b``, name_type, integ_type, sec_type])
 val term =
   Term`((M : ^(ty_antiq M_type)),(Oi : ^(ty_antiq integ_type) po),
         (Os : ^(ty_antiq sec_type) po)) sat ^f`
 in
   ASSUME term
 end;
```

See also

ACL_ASSUM2

A.2 ACL_ASSUM2

```
ACL_ASSUM2                    (acl_infRules)
```

ACL_ASSUM2 : term -> term -> term -> thm

Synopsis

Introduces an assumption in the access-control logic given a formula f, and partial orderings on integrity labels O_i and security labels O_s.

Description

When applied to a term f, which must have type Form, O_i of type `integ_type po`, and O_s of type `sec_type po`, ACL_ASSUMs introduces a theorem `(M,Oi,Os) sat f |- (M,Oi,Os) sat f`.

```
----------------------------------  ACL_ASSUM2 f Oi Os
(M,Oi,Os) sat f |- (M,Oi,Os) sat f
```

Failure

Fails unless f has type Form, and O_i and O_s have types `integ_type po` and `sec_type po`, respectively.

Example

The following application of `ACL_ASSUM2`:

```
- ACL_ASSUM2
   ``(Token:'c Princ) says (Role says f:('a,'c,'d,'e)Form)``
   ``Int_order:'d po``
   ``Sec_Order:'e po``;
```

yields the following result:

```
> val it =
   [.] |- (M,Int_order,Sec_Order) sat Token says Role says f : thm
```

Implementation

```
fun ACL_ASSUM2 f Oi Os =
let
 val f_type = type_of f
 val f_type_parts = dest_type f_type
 val [prop_type, name_type, integ_type, sec_type] = snd f_type_parts
 val M_type =
   mk_type ("Kripke",
            [prop_type, ``:'b``, name_type, integ_type, sec_type])
 val term =
   Term`((M : ^(ty_antiq M_type)),(^Oi : ^(ty_antiq integ_type) po),
         (^Os : ^(ty_antiq sec_type) po)) sat ^f`
 in
   ASSUME term
 end;
```

See also

`ACL_ASSUM`

A.3 ACL_CONJ

ACL_CONJ (acl_infRules)

```
ACL_CONJ : thm -> thm -> thm
```

Synopsis

Introduces a conjunction in the access-control logic.

Description

```
A1 |- (M,Oi,Os) sat f1   A2 |- (M,Oi,Os) sat f2
------------------------------------------------- ACL_CONJ
        A1 u A2 |- (M,Oi,Os) sat f1 andf f2
```

Failure

Fails unless both theorems are of the form `A |- (M,Oi,Os) sat f` and their types are the same.

Example

The following example shows the conjunction of two theorems.

```
- val th1 = ACL_ASSUM ``p:('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat p : thm
- val th2 = ACL_ASSUM ``q:('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat q : thm
- ACL_CONJ th1 th2;
> val it =  [..] |- (M,Oi,Os) sat p andf q : thm
```

Implementation

```
fun ACL_CONJ th1 th2 =
   MATCH_MP(MATCH_MP (SPEC_ALL Conjunction) th1) th2;
```

See also

ACL_SIMP1, ACL_SIMP2

A.4 ACL_DISJ1

```
ACL_DISJ1                    (acl_infRules)
```

`ACL_DISJ1 : term -> thm -> thm`

Synopsis

Introduces a right disjunct into the conclusion of an access-control logic theorem.

Description

```
     A |- (M,Oi,Os) sat f1
  ----------------------------  ACL_DISJ1 f2
  A |- (M,Oi,Os) sat f1 orf f2
```

Failure

Fails unless the input theorem is a disjunction in the access-control logic and the types of f1 and f2 are the same.

Example

The following introduces a *right* disjunct *q* to a theorem `[.]  |- (M,Oi,Os) sat p`.

```
- val th = ACL_ASSUM ``p:('propVar,'pName,'Int,'Sec)Form``;
> val th =  [.] |- (M,Oi,Os) sat p : thm
- ACL_DISJ1 ``q:('propVar,'pName,'Int,'Sec)Form`` th;
> val it =  [.] |- (M,Oi,Os) sat p orf q : thm
```

Implementation

```
fun ACL_DISJ1 f th =
let
 val f_type = type_of f
 val term = Term`f2:^(ty_antiq f_type)`
in
 SPEC f (GEN term (MATCH_MP (SPEC_ALL Disjunction1) th))
end;
```

See also

`ACL_DISJ2`

A.5 ACL_DISJ2

```
ACL_DISJ2                    (acl_infRules)
```

`ACL_DISJ2 : term -> thm -> thm`

Synopsis

Introduces a left disjunct into the conclusion of an access-control logic theorem.

Description

```
    A |- (M,Oi,Os) sat f2
 ----------------------------  ACL_DISJ2 f1
 A |- (M,Oi,Os) sat f1 orf f2
```

Failure

Fails unless the input theorem is a disjunction in the access-control logic and the types of f1 and f2 are the same.

Example

The following introduces a *left* disjunct *q* to a theorem `[.]  |- (M,Oi,Os) sat p`.

```
> val th =  [.] |- (M,Oi,Os) sat p : thm
- ACL_DISJ2 ``q:('propVar,'pName,'Int,'Sec)Form`` th;
> val it =  [.] |- (M,Oi,Os) sat q orf p : thm
```

Implementation

```
fun ACL_DISJ2 f th =
let
 val f_type = type_of f
 val term = Term`f1:^(ty_antiq f_type)`
in
 SPEC f (GEN term (MATCH_MP (SPEC_ALL Disjunction2) th))
end;
```

See also

ACL_DISJ1

A.6 ACL_DN

```
ACL_DN                                    (acl_infRules)
```

```
ACL_DN : thm -> thm
```

Synopsis

Applies double negation to a theorem in the access-control logic.

DESCRIPTION

```
A |- (M,Oi,Os) sat notf(notf f)
------------------------------- ACL_DN
      A |- (M,Oi,Os) sat f
```

Failure

Fails unless the input theorem is a double negation in the access-control logic.

Example

The following example shows the double negation being removed from $\neg\neg(K_{Alice} \Rightarrow Alice$.

```
- val th =
  ACL_ASSUM
  ``(notf (notf
   (K_Alice speaksfor Alice))):('propVar, 'pName, 'Int, 'Sec)Form``;
<<HOL message: inventing new type variable names: 'a, 'b>>
> val th =
   [.] |- (M,Oi,Os) sat notf (notf (K_Alice speaksfor Alice)) : thm
- ACL_DN th;
> val it =  [.] |- (M,Oi,Os) sat K_Alice speaksfor Alice : thm
```

Implementation

```
fun ACL_DN th = MATCH_MP (SPEC_ALL Double_Negation) th;
```

A.7 ACL_MP

```
ACL_MP                              (acl_infRules)
```

`ACL_MP : thm -> thm -> thm`

Synopsis

Implements Modus Ponens in the access-control logic.

Description

When applied to theorems `A1 |- (M,Oi,Os) sat f1` and `A2 |- (M,Oi,Os) sat f1 impf f2` in the access-control logic, `ACL_MP` introduces a theorem `A1 u A2 |- (M,Oi,Os) sat f2`.

```
 A1 |- (M,Oi,Os) sat f1   A2 |- (M,Oi,Os) sat f1 impf f2
 -------------------------------------------------------  ACL_MP
                 A1 u A2 |- (M,Oi,Os) sat f2
```

Failure

Fails unless f1 in the first theorem is the same as f1 in the second theorem.

Example

The following illustrates the application of `ACL_MP` to two theorems, `th1` and `th2`.

```
- val th1 = ACL_ASSUM ``p:('a,'c,'d,'e)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat p : thm
- val th2 = ACL_ASSUM ``(p impf q):('a,'c,'d,'e)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat p impf q : thm
- ACL_MP th1 th2;
> val it =  [..] |- (M,Oi,Os) sat q : thm
```

Implementation

```
fun ACL_MP th1 th2 =
    MATCH_MP (MATCH_MP (SPEC_ALL Modus_Ponens) th1) th2;
```

See also

`MP_SAYS, ACL_MT`

A.8 ACL_MT

ACL_MT	(acl_infRules)

`ACL_MT : thm -> thm -> thm`

Synopsis

Implements Modus Tollens in the access-control logic.

Description

When applied to theorems `A1 |- (M,Oi,Os) sat notf f2` and `A2 |- (M,Oi,Os) sat f1 impf f2` in the access-control logic, `ACL_MT` introduces a theorem `A1 u A2 |- (M,Oi,Os) sat notf f1`.

```
A1 |- (M,Oi,Os) sat f1 impf f2
A2 |- (M,Oi,Os) sat notf f2
-------------------------------- ACL_MT
A1 u A2 |- (M,Oi,Os) sat notf f1
```

Failure

Fails unless f_2 in the first theorem is the same as f_2 in the second theorem and the types are consistent.

Example

In the following example, `th1` corresponds to $p \supset q$ in the access-control logic, and `th2` corresponds to $\neg q$. By `ACL_MT` we can conclude $\neg p$.

```
- val th1 = ACL_ASSUM ``(p impf q):('propvar,'pname,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat p impf q : thm
- val th2 = ACL_ASSUM ``(notf q):('propvar,'pname,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat notf q : thm
- ACL_MT th1 th2;
> val it =  [..] |- (M,Oi,Os) sat notf p : thm
```

Implementation

```
fun ACL_MT th1 th2 =
  MATCH_MP (MATCH_MP (SPEC_ALL Modus_Tollens) th1) th2;
```

See also

`ACL_MP`

A.9 ACL_SIMP1

ACL_SIMP1 (acl_infRules)

ACL_SIMP1 : thm -> thm

Synopsis
Extracts left conjunct of a theorem in the access-control logic.

Description

```
A |- (M,Oi,Os) sat f1 andf f2
----------------------------- ACL_SIMP1
     A |- (M,Oi,Os) sat f1
```

Failure
Fails unless the input theorem is a conjunction in the access-control logic.

Example
The following example shows the application of ACL_SIMP1 to the theorem $(M, O_i, O_s) \models p \wedge q$ in the access-control logic.

```
- val th =
    ACL_ASSUM ``(p andf q):('propvar,'princName,'Int,'Sec)Form``;
> val th =  [.] |- (M,Oi,Os) sat p andf q : thm
- ACL_SIMP1 th;
> val it =  [.] |- (M,Oi,Os) sat p : thm
```

Implementation

```
fun ACL_SIMP1 th = MATCH_MP (SPEC_ALL Simplification1) th;
```

See also
ACL_SIMP2

A.10 ACL_SIMP2

```
ACL_SIMP2                    (acl_infRules)
```

```
ACL_SIMP2 : thm -> thm
```

Synopsis
Extracts left conjunct of a theorem in the access-control logic.

Description

```
A |- (M,Oi,Os) sat f1 andf f2
----------------------------- ACL_SIMP2
     A |- (M,Oi,Os) sat f2
```

Failure
Fails unless the input theorem is a conjunction in the access-control logic.

Example
The following example shows the application of `ACL_SIMP2` to the theorem $(M, O_i, O_s) \models p \wedge q$ in the access-control logic.

```
- val th =
    ACL_ASSUM ``(p andf q):('propvar,'princName,'Int,'Sec)Form``;
> val th =  [.] |- (M,Oi,Os) sat p andf q : thm
- ACL_SIMP2 th;
> val it =  [.] |- (M,Oi,Os) sat q : thm
```

Implementation

```
fun ACL_SIMP2 th = MATCH_MP (SPEC_ALL Simplification2) th;
```

See also
`ACL_SIMP1`

A.11 ACL_TAUT

ACL_TAUT	(acl_infRules)

`ACL_TAUT : term -> thm`

Synopsis

Attempts to prove a proposition f in the access-control logic is true in all Kripke models (M, O_i, O_s).

Description

When applied to a term f, which must have type Form, `ACL_TAUT` attempts to prove `(M,Oi,Os) sat f`.

```
------------------  ACL_TAUT f
|- (M,Oi,Os) sat f
```

Failure

Fails if f is not a tautology.

Example

The application of `ACL_TAUT` as shown below

```
- ACL_TAUT ``(p orf notf p):('a,'c,'d,'e)Form``;
```

yields the result

```
> val it = |- (M,Oi,Os) sat p orf notf p : thm
```

Implementation

```
fun ACL_TAUT f =
let
 val f_type = type_of f
 val f_type_parts = dest_type f_type
 val [prop_type, name_type, integ_type, sec_type] = snd f_type_parts
 val M_type =
   mk_type ("Kripke",
           [prop_type, ``:'b``, name_type, integ_type, sec_type])
 val term =
   Term`((M : ^(ty_antiq M_type)),(Oi : ^(ty_antiq integ_type) po),
         (Os : ^(ty_antiq sec_type) po)) sat ^f`
 in
    TAC_PROOF(([],term),ACL_TAUT_TAC)
 end;
```

See also

ACL_TAUT_TAC

A.12 ACL_TAUT_TAC

```
ACL_TAUT_TAC                    (acl_infRules)
```

ACL_TAUT_TAC : tactic

Synopsis

Invoke decision procedures to prove propositional formulas and partial order relations in the access-control logic.

Description

When given a propositional formula f in the access-control logic using only notf, andf, orf, impf, eqf, eqn, lte, and lt, ACL_TAUT_TAC attempts to prove f true in all Kripke structures (M, O_i, O_s).

```
 A ?- (M,Oi,Os) sat f
====================== ACL_TAUT_TAC
 A |-  (M,Oi,Os) sat f
```

Failure

Fails if f is not a propositional tautology, e.g., p and notf p.

Example
The use of ACL_TAUT_TAC within the following proof

```
- TAC_PROOF
  (([],``(M:('a,'b,'c,'d,'e)Kripke,Int_Order:'d po, Sec_Order:'e po)
          sat (p orf notf p):('a,'c,'d,'e)Form``),
         ACL_TAUT_TAC);
```

yields the following result

```
> val it = |- (M,Int_Order,Sec_Order) sat p orf notf p : thm
```

Implementation

```
val ACL_TAUT_TAC =
    REWRITE_TAC
    [sat_allworld, world_T, world_F, world_not,
     world_and, world_or, world_imp, world_eq,
     world_eqn, world_lte, world_lt]
    THEN DECIDE_TAC;
```

See also
ACL_TAUT

A.13 AND_SAYS_RL

```
AND_SAYS_RL                    (acl_infRules)
```

```
AND_SAYS_RL : thm -> thm
```

Synopsis
Applies And_Says_Eq theorem to rewrite terms

Description

```
   A |- (M,Oi,Os) sat (P says f) andf (Q says f)
 --------------------------------------------------- AND_SAYS_RL
        A |- (M,Oi,Os) sat P meet Q says f
```

Failure
Fails unless the input theorem is of the form P says f andf Q says f and all types are consistent.

Example

```
- val th1 = ACL_ASSUM
   ``((P says f) andf (Q says f)):('Prop,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat P says f andf Q says f : thm
- val th2 = AND_SAYS_RL th1;
> val th2 =  [.] |- (M,Oi,Os) sat P meet Q says f : thm
```

Implementation

```
fun AND_SAYS_RL th = REWRITE_RULE [GSYM(SPEC_ALL And_Says_Eq)] th;
```

See also
AND_SAYS_LR

A.14 AND_SAYS_LR

```
AND_SAYS_LR                    (acl_infRules)
```

AND_SAYS_LR : thm -> thm

Synopsis
Applies And_Says_Eq theorem to rewrite terms

Description

```
   A |- (M,Oi,Os) sat (P meet Q says f)
 ------------------------------------------ AND_SAYS_LR
 A |- (M,Oi,Os) sat P says f andf Q says f
```

Failure
Fails unless the input theorem is of the form P meet Q says f and all types are consistent.

Example

```
- val th1 = ACL_ASSUM
   ``(P meet Q says f):('Prop,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat P meet Q says f : thm
- val th2 = AND_SAYS_LR th1;
> val th2 =  [.] |- (M,Oi,Os) sat P says f andf Q says f : thm
```

Implementation

```
fun AND_SAYS_LR th = REWRITE_RULE [SPEC_ALL And_Says_Eq] th;
```

See also

AND_SAYS_RL

A.15 CONTROLS

CONTROLS (acl_infRules)

CONTROLS : thm->thm -> thm

Synopsis

Deduces formula f if the principal who says f also controls f.

Description

```
A1 |- (M,Oi,Os) sat P controls f
  A2 |- (M,Oi,Os) sat P says f
-------------------------------- CONTROLS
   A1 u A2 |- (M,Oi,Os) sat f
```

Failure

Fails unless the theorems match in terms of principals and formulas in the access-control logic.

Example

The following is an example of Alice controlling and saying f.

```
- val th1 =
   ACL_ASSUM ``(Alice controls f):('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat Alice controls f : thm
- val th2 =
   ACL_ASSUM ``(Alice says f):('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat Alice says f : thm
- CONTROLS th1 th2;
> val it =  [..] |- (M,Oi,Os) sat f : thm
```

Implementation

```
fun CONTROLS th1 th2 =
   MATCH_MP (MATCH_MP (SPEC_ALL Controls) th2) th1;
```

See also

DC, REPS

A.16 DC

DC	(acl_infRules)

DC : thm -> thm -> thm

Synopsis

Applies Derived Controls rule to theorems in the access-control logic.

Description

```
    A1 |- (M,Oi,Os) sat P speaks_for Q
     A2 |- (M,Oi,Os) sat Q controls f
  ------------------------------------- DC
  A1 u A2 |- (M,Oi,Os) sat P controls f
```

Failure

Fails unless the input theorems match in their corresponding principal names.

Example

```
- val th1 =
   ACL_ASSUM
   ``(Alice speaks_for Bob):('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat Alice speaks_for Bob : thm
- val th2 =
   ACL_ASSUM ``(Bob controls f):('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat Bob controls f : thm
- DC th1 th2;
> val it =  [..] |- (M,Oi,Os) sat Alice controls f : thm
```

Implementation

```
fun DC th1 th2 =
   MATCH_MP(MATCH_MP (SPEC_ALL Derived_Controls) th1) th2;
```

See also

CONTROLS, SPEAKS_FOR

A.17 DOMI_TRANS

```
DOMI_TRANS                        (acl_infRules)
```

DOMI_TRANS : thm -> thm -> thm

Synopsis

Applies transitivity of domi to theorems in the access-control logic.

Description

```
     A1 |- (M,Oi,Os) sat l1 domi l2
     A2 |- (M,Oi,Os) sat l2 domi l3
  ----------------------------------- DOMI_TRANS
  A1 u A2 |- (M,Oi,Os) sat l1 domi l3
```

Failure

Fails unless the input theorems match in their corresponding terms and types.

Example

```
- val th1 =
   ACL_ASSUM ``(l1 domi l2):('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat l1 domi l2 : thm
- val th2 =
   ACL_ASSUM ``(l2 domi l3):('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat l2 domi l3 : thm
- DOMI_TRANS th1 th2;
> val it =  [..] |- (M,Oi,Os) sat l1 domi l3 : thm
```

Implementation

```
fun DOMI_TRANS th1 th2 =
     MATCH_MP(MATCH_MP (SPEC_ALL domi_transitive) th1) th2;
```

See also

DOMS_TRANS, IL_DOMI, SL_DOMS

A.18 DOMS_TRANS

```
DOMS_TRANS                    (acl_infRules)
```

DOMS_TRANS : thm -> thm -> thm

Synopsis

Applies transitivity of doms to theorems in the access-control logic.

Description

```
     A1 |- (M,Oi,Os) sat l1 doms l2
     A2 |- (M,Oi,Os) sat l2 doms l3
  ----------------------------------- DOMS_TRANS
  A1 u A2 |- (M,Oi,Os) sat l1 doms l3
```

Failure

Fails unless l1, l2, and l3 match appropriately and have the same type.

Example

```
- val th1 =
   ACL_ASSUM ``(l1 doms l2):('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat l1 doms l2 : thm
- val th2 =
   ACL_ASSUM ``(l2 doms l3):('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat l2 doms l3 : thm
- DOMS_TRANS th1 th2;
> val it =  [..] |- (M,Oi,Os) sat l1 doms l3 : thm
```

Implementation

```
fun DOMS_TRANS th1 th2 =
      MATCH_MP(MATCH_MP (SPEC_ALL doms_transitive) th1) th2;
```

See also

DOMI_TRANS, SL_DOMS, IL_DOMI

A.19 EQF_ANDF1

```
EQF_ANDF1                       (acl_infRules)
```

EQF_ANDF1 : thm -> thm -> thm

Synopsis

Applies eqf_andf1 to substitute an equivalent term for another in the left conjunct.

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat f andf g
---------------------------------- EQF_ANDF1
A1 u A2 |- (M,Oi,Os) sat f' andf g
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a conjunction. Fails unless all the types are consistent.

Example

```
- val EQF_ANDF1_Test =
let
 val th1 = ACL_ASSUM``(p1 eqf p1'):('a,'c,'d,'e)Form``
 val th2 = ACL_ASSUM``(p1 andf p2):('a,'c,'d,'e)Form``
 val th3 = EQF_ANDF1 th1 th2
 val th4 = DISCH (hd(hyp th2)) th3
in
 DISCH(hd(hyp th1)) th4
end;
> val EQF_ANDF1_Test =
    |- (M,Oi,Os) sat p1 eqf p1' ==>
       (M,Oi,Os) sat p1 andf p2 ==>
       (M,Oi,Os) sat p1' andf p2
     : thm
```

Implementation

```
fun EQF_ANDF1 th1 th2 =
let
 val th3 = MATCH_MP eqf_andf1 th1
in
 MATCH_MP th3 th2
end
```

See also

EQF_ANDF2

A.20 EQF_ANDF2

```
EQF_ANDF2                    (acl_infRules)
```

EQF_ANDF2 : thm -> thm -> thm

Synopsis

Applies eqf_andf2 to substitute an equivalent term for another in the right conjunct.

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat g andf f
---------------------------------- EQF_ANDF2
A1 u A2 |- (M,Oi,Os) sat g andf f'
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a conjunction. Fails unless all the types are consistent.

Example

```
- val EQF_ANDF2_Test =
let
 val th1 = ACL_ASSUM``(p2 eqf p2'):('a,'c,'d,'e)Form``
 val th2 = ACL_ASSUM``(p1 andf p2):('a,'c,'d,'e)Form``
 val th3 = EQF_ANDF2 th1 th2
 val th4 = DISCH (hd(hyp th2)) th3
in
 DISCH(hd(hyp th1)) th4
end;
> val EQF_ANDF2_Test =
    |- (M,Oi,Os) sat p2 eqf p2' ==>
       (M,Oi,Os) sat p1 andf p2 ==>
       (M,Oi,Os) sat p1 andf p2'
     : thm
```

Implementation

```
fun EQF_ANDF2 th1 th2 =
let
 val th3 = MATCH_MP eqf_andf2 th1
in
 MATCH_MP th3 th2
end
```

See also

EQF_ANDF1

A.21 EQF_CONTROLS

```
EQF_CONTROLS                    (acl_infRules)
```

```
EQF_CONTROLS : thm -> thm -> thm
```

Synopsis

Applies eqf_controls to substitute an equivalent term f' for f in the term P says f.

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat P controls f
-------------------------------------- EQF_CONTROLS
A1 u A2 |- (M,Oi,Os) sat P controls f'
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a conjunction. Fails unless all the types are consistent.

Example

```
- val EQF_CONTROLS_Test =
let
  val th1 = ACL_ASSUM``(f eqf f'):('a,'c,'d,'e)Form``
  val th2 = ACL_ASSUM``(P controls f):('a,'c,'d,'e)Form``
  val th3 = EQF_CONTROLS th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;
> val EQF_CONTROLS_Test =
    |- (M,Oi,Os) sat f eqf f' ==>
       (M,Oi,Os) sat P controls f ==>
       (M,Oi,Os) sat P controls f'
     : thm
```

Implementation

```
fun EQF_CONTROLS th1 th2 =
let
 val th3 = MATCH_MP eqf_controls th1
in
 MATCH_MP th3 th2
end
```

See also

EQF_SAYS

A.22 EQF_EQF1

EQF_EQF1	(acl_infRules)

```
EQF_EQF1 : thm -> thm -> thm
```

Synopsis

Applies eqf_controls to substitute f' for f in the term $f \equiv g$.

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat f eqf g
--------------------------------- EQF_EQF1
A1 u A2 |- (M,Oi,Os) sat f' eqf g
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a equivalence. Fails unless all the types are consistent.

Example

```
- val EQF_EQF1_Test =
let
  val th1 = ACL_ASSUM``(f eqf f'):('a,'c,'d,'e)Form``
  val th2 = ACL_ASSUM``(f eqf g):('a,'c,'d,'e)Form``
  val th3 = EQF_EQF1 th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;
> val EQF_EQF1_Test =
     |- (M,Oi,Os) sat f eqf f' ==>
        (M,Oi,Os) sat f eqf g ==>
        (M,Oi,Os) sat f' eqf g
      : thm
```

Implementation

```
fun EQF_EQF1 th1 th2 =
let
 val th3 = MATCH_MP eqf_eqf1 th1
in
 MATCH_MP th3 th2
end
```

See also
EQF_EQF2

A.23 EQF_EQF2

```
EQF_EQF2                          (acl_infRules)
```

EQF_EQF2 : thm -> thm -> thm

Synopsis
Applies eqf_controls to substitute f' for f in the term $f \equiv g$.

Description

```
   A1 |- (M,Oi,Os) sat f eqf f’
   A2 |- (M,Oi,Os) sat g eqf f
---------------------------------- EQF_EQF2
A1 u A2 |- (M,Oi,Os) sat g eqf f’
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a equivalence. Fails unless all the types are consistent.

Example

```
- val EQF_EQF2_Test =
let
  val th1 = ACL_ASSUM``(f eqf f’):(’a,’c,’d,’e)Form``
  val th2 = ACL_ASSUM``(g eqf f):(’a,’c,’d,’e)Form``
  val th3 = EQF_EQF2 th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;
> val EQF_EQF2_Test =
    |- (M,Oi,Os) sat f eqf f’ ==>
       (M,Oi,Os) sat g eqf f ==>
       (M,Oi,Os) sat g eqf f’
     : thm
```

Implementation

```
fun EQF_EQF2 th1 th2 =
let
 val th3 = MATCH_MP eqf_eqf2 th1
in
 MATCH_MP th3 th2
end
end
```

See also

EQF_EQF1

A.24 EQF_IMPF1

```
EQF_IMPF1                    (acl_infRules)
```

```
EQN_IMPF1 : thm -> thm -> thm
```

Synopsis

Applies eqf_impf1 to substitute an equivalent term for another in the left side of an implication.

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat f impf g
---------------------------------- EQF_IMPF1
A1 u A2 |- (M,Oi,Os) sat f' impf g
```

Example

```
- val EQF_IMPF1_Test =
let
  val th1 = ACL_ASSUM``(f eqf f'):('a,'c,'d,'e)Form``
  val th2 = ACL_ASSUM``(f impf g):('a,'c,'d,'e)Form``
  val th3 = EQF_IMPF1 th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;
> val EQF_IMPF1_Test =
    |- (M,Oi,Os) sat f eqf f' ==>
       (M,Oi,Os) sat f impf g ==>
       (M,Oi,Os) sat f' impf g
     : thm
```

Implementation

```
fun EQF_IMPF1 th1 th2 =
let
 val th3 = MATCH_MP eqf_impf1 th1
in
 MATCH_MP th3 th2
end
```

See also
EQF_IMPF2

A.25 EQF_IMPF2

EQF_IMPF2	(acl_infrules)

```
EQF_IMPF2 : thm -> thm -> thm
```

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat g impf f
---------------------------------- EQF_IMPF2
A1 u A2 |- (M,Oi,Os) sat g impf f'
```

Failure
Fails unless the first theorem is an equivance and the second theorem is an implication. Fails unless all the types are consistent.

Example

```
- val EQF_IMPF2_Test =
let
  val th1 = ACL_ASSUM``(f eqf f'):('a,'c,'d,'e)Form``
  val th2 = ACL_ASSUM``(g impf f):('a,'c,'d,'e)Form``
  val th3 = EQF_IMPF2 th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;
> val EQF_IMPF2_Test =
     |- (M,Oi,Os) sat f eqf f' ==>
        (M,Oi,Os) sat g impf f ==>
        (M,Oi,Os) sat g impf f'
      : thm
```

Implementation

```
fun EQF_IMPF2 th1 th2 =
let
 val th3 = MATCH_MP eqf_impf2 th1
in
 MATCH_MP th3 th2
end
```

See also

EQF_IMPF1

A.26 EQF_NOTF

EQF_NOTF	(acl_infrules)

```
EQF_NOTF : thm -> thm -> thm
```

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat notf f
---------------------------------- EQF_NOTF
A1 u A2 |- (M,Oi,Os) sat notf f'
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a negation. Fails unless all the types are consistent.

Example

```
- val EQF_NOTF_Test =
let
  val th1 = ACL_ASSUM``(f eqf f'):('a,'c,'d,'e)Form``
  val th2 = ACL_ASSUM``(notf f):('a,'c,'d,'e)Form``
  val th3 = EQF_NOTF th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;;
> val EQF_NOTF_Test =
     |- (M,Oi,Os) sat f eqf f' ==>
        (M,Oi,Os) sat notf f ==>
        (M,Oi,Os) sat notf f'
      : thm
```

Implementation

```
fun EQF_NOTF th1 th2 =
let
 val th3 = MATCH_MP eqf_notf th1
in
 MATCH_MP th3 th2
end
```

A.27 EQF_ORF1

```
EQF_ORF1                              (acl_infrules)
```

`EQF_ORF1 : thm -> thm -> thm`

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat f orf g
--------------------------------- EQF_ORF1
A1 u A2 |- (M,Oi,Os) sat f' orf g
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a disjunction. Fails unless all the types are consistent.

Example

```
- val EQF_ORF1_Test =
let
  val th1 = ACL_ASSUM``(f eqf f'):('a,'c,'d,'e)Form``
  val th2 = ACL_ASSUM``(f orf g):('a,'c,'d,'e)Form``
  val th3 = EQF_ORF1 th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;
> val EQF_ORF1_Test =
     |- (M,Oi,Os) sat f eqf f' ==>
        (M,Oi,Os) sat f orf g ==>
        (M,Oi,Os) sat f' orf g
      : thm
```

Implementation

```
fun EQF_ORF1 th1 th2 =
let
 val th3 = MATCH_MP eqf_orf1 th1
in
 MATCH_MP th3 th2
end
```

See also

EQF_ORF2

A.28 EQF_ORF2

EQF_ORF2	(acl_infrules)

EQF_ORF2 : thm -> thm -> thm

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat g orf f
--------------------------------- EQF_ORF2
A1 u A2 |- (M,Oi,Os) sat g orf f'
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a disjunction. Fails unless all the types are consistent.

Example

```
- val EQF_ORF2_Test =
let
  val th1 = ACL_ASSUM``(f eqf f'):('a,'c,'d,'e)Form``
  val th2 = ACL_ASSUM``(g orf f):('a,'c,'d,'e)Form``
  val th3 = EQF_ORF2 th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;
> val EQF_ORF2_Test =
     |- (M,Oi,Os) sat f eqf f' ==>
        (M,Oi,Os) sat g orf f ==>
        (M,Oi,Os) sat g orf f'
      : thm
```

Implementation

```
fun EQF_ORF2 th1 th2 =
let
 val th3 = MATCH_MP eqf_orf2 th1
in
 MATCH_MP th3 th2
end
```

See also

EQF_ORF1

A.29 EQF_REPS

```
EQF_REPS                          (acl_infrules)
```

`EQF_REPS : thm -> thm -> thm`

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat reps P Q f
------------------------------------ EQF_REPS
A1 u A2 |- (M,Oi,Os) sat reps P Q f'
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a delegation. Fails unless all the types are consistent.

Example

```
- val EQF_REPS_Test =
let
  val th1 = ACL_ASSUM``(f eqf f'):('a,'c,'d,'e)Form``
  val th2 = ACL_ASSUM``(reps P Q f):('a,'c,'d,'e)Form``
  val th3 = EQF_REPS th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;
> val EQF_REPS_Test =
    |- (M,Oi,Os) sat f eqf f' ==>
       (M,Oi,Os) sat reps P Q f ==>
       (M,Oi,Os) sat reps P Q f'
     : thm
```

Implementation

```
fun EQF_REPS th1 th2 =
let
 val th3 = MATCH_MP eqf_reps th1
in
 MATCH_MP th3 th2
end
```

A.30 EQF_SAYS

```
EQF_SAYS                    (acl_infrules)
```

```
EQF_SAYS : thm -> thm -> thm
```

Description

```
   A1 |- (M,Oi,Os) sat f eqf f'
   A2 |- (M,Oi,Os) sat P says f
----------------------------------- EQF_SAYS
A1 u A2 |- (M,Oi,Os) sat P says f'
```

Failure

Fails unless the first theorem is an equivance and the second theorem is a says statement. Fails unless all the types are consistent.

Example

```
- val EQF_SAYS_Test =
let
  val th1 = ACL_ASSUM``(f eqf f'):('a,'c,'d,'e)Form``
  val th2 = ACL_ASSUM``(P says f):('a,'c,'d,'e)Form``
  val th3 = EQF_SAYS th1 th2
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;
> val EQF_SAYS_Test =
    |- (M,Oi,Os) sat f eqf f' ==>
       (M,Oi,Os) sat P says f ==>
       (M,Oi,Os) sat P says f'
     : thm
```

Implementation

```
fun EQF_SAYS th1 th2 =
let
 val th3 = MATCH_MP eqf_says th1
in
 MATCH_MP th3 th2
end
```

A.31 EQN_EQN

```
EQN_EQN                         (acl_infRules)
```

```
EQN_EQN : thm -> thm -> thm -> thm
```

Synopsis
Applies eqn_eqn to theorems in the access-control logic.

Description
```
        A1 |- (M,Oi,Os) sat c1 eqn n1
        A2 |- (M,Oi,Os) sat c2 eqn n2
        A3 |- (M,Oi,Os) sat n1 eqn n2
 --------------------------------------- EQN_EQN
 A1 u A2 u A3 |- (M,Oi,Os) sat c1 eqn c2
```

Failure
Fails unless the types are consistent among the three theorems.

Example

```
- val th1 = ACL_ASSUM ``c1 eqn n1``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
> val th1 =  [.] |- (M,Oi,Os) sat c1 eqn n1 : thm
- val th2 = ACL_ASSUM ``c2 eqn n2``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
> val th2 =  [.] |- (M,Oi,Os) sat c2 eqn n2 : thm
- val th3 = ACL_ASSUM ``n1 eqn n2``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
> val th3 =  [.] |- (M,Oi,Os) sat n1 eqn n2 : thm
- EQN_EQN th1 th2 th3;
> val it =  [...] |- (M,Oi,Os) sat c1 eqn c2 : thm
```

Implementation

```
fun EQN_EQN th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP eqn_eqn th1) th2) th3;
```

See also
EQN_LT, EQN_LTE

A.32 EQN_LT

EQN_LT	(acl_infRules)

EQN_LT : thm -> thm -> thm -> thm

Synopsis

Applies eqn_lt to theorems in the access-control logic.

Description

```
         A1 |- (M,Oi,Os) sat c1 eqn n1
         A2 |- (M,Oi,Os) sat c2 eqn n2
         A3 |- (M,Oi,Os) sat n1 lt n2
  -------------------------------------------- EQN_LT
  A1 u A2 u A3 |- (M,Oi,Os) sat c1 lt c2
```

Failure

Fails unless the types are consistent among the three theorems.

Example

```
- val th1 = ACL_ASSUM ``c1 eqn n1``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
> val th1 =  [.] |- (M,Oi,Os) sat c1 eqn n1 : thm
- val th2 = ACL_ASSUM ``c2 eqn n2``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
> val th2 =  [.] |- (M,Oi,Os) sat c2 eqn n2 : thm
- val th3 = ACL_ASSUM ``n1 lt n2``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
> val th3 =  [.] |- (M,Oi,Os) sat n1 lt n2 : thm
- EQN_LT th1 th2 th3;
> val it =  [...] |- (M,Oi,Os) sat c1 lt c2 : thm
```

Implementation

```
fun EQN_LT th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP eqn_lt th1) th2) th3;
```

See also

EQN_LTE, EQN_EQN

A.33 EQN_LTE

EQN_LTE	(acl_infRules)

```
EQN_LTE : thm -> thm -> thm -> thm
```

Synopsis

Applies eqn_lte to theorems in the access-control logic

Description

```
       A1 |- (M,Oi,Os) sat c1 eqn n1
       A2 |- (M,Oi,Os) sat c2 eqn n2
       A3 |- (M,Oi,Os) sat n1 lte n2
-------------------------------------------- EQN_LTE
A1 u A2 u A3 |- (M,Oi,Os) sat c1 lte c2
```

Failure

Fails unless the types are consistent amount the three theorems.

Example

```
- val th1 = ACL_ASSUM ``c1 eqn n1``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
> val th1 =  [.] |- (M,Oi,Os) sat c1 eqn n1 : thm
- val th2 = ACL_ASSUM ``c2 eqn n2``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
> val th2 =  [.] |- (M,Oi,Os) sat c2 eqn n2 : thm
- val th3 = ACL_ASSUM ``n1 lte n2``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
> val th3 =  [.] |- (M,Oi,Os) sat n1 lte n2 : thm
- EQN_LTE th1 th2 th3;
> val it =  [...] |- (M,Oi,Os) sat c1 lte c2 : thm
```

Implementation

```
fun EQN_LTE th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP eqn_lte th1) th2) th3;
```

See also

EQN_LT, EQN_EQN

A.34 HS

```
HS                                    (acl_infRules)
```

`HS : thm -> thm -> thm`

Synopsis

Applies hypothetical syllogism to theorems in the access-control logic.

Description

```
     A1 |- (M,Oi,Os) sat f1 impf f2
     A2 |- (M,Oi,Os) sat f2 impf f3
  ----------------------------------- HS
  A1 u A2 |- (M,Oi,Os) sat f1 impf f3
```

Failure

Fails unless the input theorems match in their consequent and antecedent in the access-control logic, and their types are the same.

Example

The following is an example of hypothetical syllogism applied to $p \supset q$ and $q \supset r$.

```
- val th1 =
   ACL_ASSUM ``(p impf q):('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat p impf q : thm
- val th2 =
   ACL_ASSUM ``(q impf r):('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat q impf r : thm
- HS th1 th2;
> val it =  [..] |- (M,Oi,Os) sat p impf r : thm
```

Implementation

```
fun HS th1 th2 =
   MATCH_MP(MATCH_MP (SPEC_ALL Hypothetical_Syllogism) th1) th2;
```

See also

ACL_MP, ACL_MT, MP_SAYS

A.35 IDEMP_SPEAKS_FOR

```
IDEMP_SPEAKS_FOR        (acl_infRules)
```

```
IDEMP_SPEAKS_FOR : term -> thm
```

Synopsis

Specializes Idemp_Speaks_For to principal P

Description

```
 -------------------------------- IDEMP_SPEAKS_FOR P
  |- (M,Oi,Os) sat P speaks_for P
```

Failure

Fails unless the term is a principal

Example

Introduces `P speaks_for P`. We can change the type variables with INST_TYPE.

```
- val th1 = IDEMP_SPEAKS_FOR ``P:'pName Princ``;
> val th1 =
     []
    |- ((M :('a, 'b, 'pName, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((P :'pName Princ) speaks_for P) :('a, 'pName, 'd, 'e) Form) : thm
- val th2 =
   INST_TYPE [``:'a`` |-> ``:'Prop``, ``:'d`` |-> ``:'Int``, ``:'e`` |-> ``:'Sec`` ] th1;

> val th2 =
     []
    |- ((M :('Prop, 'b, 'pName, 'Int, 'Sec) Kripke),(Oi :'Int po),
        (Os :'Sec po)) sat
       (((P :'pName Princ) speaks_for P) :('Prop, 'pName, 'Int, 'Sec) Form) :
  thm
```

Implementation

```
fun IDEMP_SPEAKS_FOR term =
  ISPEC term (GEN ``P:'c Princ``(SPEC_ALL Idemp_Speaks_For));
```

See also

SPEAKS_FOR, MONO_SPEAKS_FOR

A.36 IL_DOMI

```
IL_DOMI                         (acl_infRules)
```

```
IL_DOMI : thm -> thm -> thm -> thm
```

Synopsis

Applies *il_domi* to theorems in the access-control logic.

Description

```
         A1 |- (M,Oi,Os) sat il P eqi l1
         A2 |- (M,Oi,Os) sat il Q eqi l2
         A3 |- (M,Oi,Os) sat l2 domi l1
  -------------------------------------------- IL_DOMI
  A1 u A2 u A3 |- (M,Oi,Os) sat il Q domi il P
```

Failure

Fails unless the types are consistent among the three theorems.

Example

```
val th1 =
   ACL_ASSUM
   ``((il Alice) eqi l1):('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat il Alice eqi l1 : thm
- val th2 =
   ACL_ASSUM ``((il Bob) eqi l2):('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat il Bob eqi l2 : thm
- val levelsRelation =
   ACL_ASSUM ``(l2 domi l1):('propVar,'pName,'Int,'Sec)Form``;
> val levelsRelation =  [.] |- (M,Oi,Os) sat l2 domi l1 : thm
- IL_DOMI th1 th2 levelsRelation;
> val it =  [...] |- (M,Oi,Os) sat il Bob domi il Alice : thm
```

Implementation

```
fun IL_DOMI th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP il_domi th1) th2) th3;
```

See also

DOMI_TRANS, DOMS_TRANS, SL_DOMS

A.37 MONO_SPEAKS_FOR

```
MONO_SPEAKS_FOR          (acl_infRules)
```

```
MONO_SPEAKS_FOR : thm -> thm -> thm
```

Synopsis

Applies Mono_speaks_for to theorems in the access-control logic

Description

```
     A1 |- (M,Oi,Os) sat P speaks_for P'
     A2 |- (M,Oi,Os) sat Q speaks_for Q'
 ------------------------------------------------------------------ MONO_SPEAKS_FOR
  A1 u A2 |- (M,Oi,Os) sat (P quoting Q) speaks_for (P' quoting Q')
```

Failure

Fails unless the types are consistent among the two theorems.

Example

```
- val th1 = ACL_ASSUM ``(P speaks_for P'):('Prop,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat P speaks_for P' : thm
- val th2 = ACL_ASSUM ``(Q speaks_for Q'):('Prop,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat Q speaks_for Q' : thm
- MONO_SPEAKS_FOR th1 th1;
> val it =  [.] |- (M,Oi,Os) sat P quoting P speaks_for P' quoting P' : thm
```

Implementation

```
fun MONO_SPEAKS_FOR th1 th2 =
    (MATCH_MP (MATCH_MP Mono_speaks_for th1) th2);
```

See also

SPEAKS_FOR, IDEMP_SPEAKS_FOR, TRANS_SPEAKS_FOR

A.38 MP_SAYS

```
MP_SAYS                         (acl_infRules)
```

```
MP_SAYS : term -> term -> term -> thm
```

Synopsis

Implements the *MP Says* rule.

Description

```
 ------------------------------------ MP_SAYS P f1 f2
 |- (M,Oi,Os) sat
     (P says (f1 impf f2)) impf
      ((P says f1) impf (P says f2))
```

Failure

Fails unless *princ* is a principal, f_1 and f_2 are terms in the access-control logic, and the types of *princ*, f_1, and f_2 are all consistent.

Example

```
- MP_SAYS ``Alice:'name Princ`` ``p:('propvar,'name,'Int,'Sec)Form``
          ``q:('propvar,'name,'Int,'Sec)Form``;
> val it =
  |- (M,Oi,Os) sat
     Alice says (p impf q) impf Alice says p impf Alice says q : thm
```

Implementation

```
fun MP_SAYS princ f1 f2 =
let
 val f1_type = type_of f1
 val f1_type_parts = dest_type f1_type
 val [prop_type, name_type, integ_type, sec_type] =
     snd f1_type_parts
 val M_type =
   mk_type ("Kripke",
            [prop_type, ``:'b``, name_type, integ_type, sec_type])
 in
   ISPECL
   [``M : ^(ty_antiq M_type)``,``Oi : ^(ty_antiq integ_type) po``,
    ``Os : ^(ty_antiq sec_type) po``, princ, f1, f2]
   MP_Says
 end;
```

See also

ACL_MP, SAYS

A.39 QUOTING_LR

QUOTING_LR	(acl_infRules)

```
QUOTING_LR : thm -> thm
```

Synopsis

Applies quoting rule to theorems in the access-control logic.

Description

```
 th [P quoting Q says f/A]
 ------------------------- QUOTING_LR
   th [P says Q says f/A]
```

Failure

Fails unless the input theorem the access-control logic.

Example

```
- val th =
   ACL_ASSUM
   ``(Alice quoting Bob says f):('propVar,'pName,'Int,'Sec)Form``;
> val th =  [.] |- (M,Oi,Os) sat Alice quoting Bob says f : thm
- QUOTING_LR th;
> val it =  [.] |- (M,Oi,Os) sat Alice says Bob says f : thm
```

Implementation

```
fun QUOTING_LR th = REWRITE_RULE [SPEC_ALL Quoting_Eq] th;
```

See also

QUOTING_RL

A.40 QUOTING_RL

```
QUOTING_RL                    (acl_infRules)
```

QUOTING_RL : thm -> thm

Synopsis

Applies quoting rule to theorems in the access-control logic.

Description

```
 th [P says Q says f/A]
------------------------- QUOTING_RL
th [P quoting Q says f/A]
```

Failure

Fails unless the input theorem the access-control logic.

Example

```
- val th =
   ACL_ASSUM
   ``(Alice says Bob says f):('propVar,'pName,'Int,'Sec)Form``;
> val th =  [.] |- (M,Oi,Os) sat Alice says Bob says f : thm
- QUOTING_RL th;
> val it =  [.] |- (M,Oi,Os) sat Alice quoting Bob says f : thm
```

Implementation

```
fun QUOTING_RL th = REWRITE_RULE [GSYM(SPEC_ALL Quoting_Eq)] th;
```

See also

QUOTING_LR

A.41 REPS

```
REPS                              (acl_infRules)
```

REPS : thm -> thm -> thm -> thm

Synopsis

Concludes statement f given theorems on delegation, quoting, and jurisdiction.

Description

```
     A1 |- (M,Oi,Os) sat reps P Q f
A2 |- (M,Oi,Os) sat (P quoting Q) says f
    A3 |- (M,Oi,Os) sat Q controls f
----------------------------------------- REPS
    A1 u A2 u A3 |- (M,Oi,Os) sat f
```

Failure

Fails unless M, Oi, Os, P, Q, and f match in all three theorems and their types are the same.

Example

The following example shows Alice as Bob's delegate requesting f on Bob's behalf and deriving f based on the `REPS` rule.

```
- val th1 =
   ACL_ASSUM ``(reps Alice Bob f):('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat reps Alice Bob f : thm
- val th2 =
   ACL_ASSUM
   ``((Alice quoting Bob) says f):('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat Alice quoting Bob says f : thm
- val th3 =
   ACL_ASSUM ``(Bob controls f):('propVar,'pName,'Int,'Sec)Form``;
> val th3 =  [.] |- (M,Oi,Os) sat Bob controls f : thm
- REPS th1 th2 th3;
> val it =  [...] |- (M,Oi,Os) sat f : thm
```

Implementation

```
fun REPS th1 th2 th3 =
    MATCH_MP(MATCH_MP (MATCH_MP (SPEC_ALL Reps) th1) th2) th3;
```

See also

REP_SAYS

A.42 REP_SAYS

```
REP_SAYS                                      (acl_infRules)
```

REP_SAYS : thm -> thm -> thm

Synopsis

Concludes statement f given theorems on delegation, quoting, and jurisdiction.

Description

```
     A1 |- (M,Oi,Os) sat reps P Q f
  A2 |- (M,Oi,Os) sat (P quoting Q) says f
  ---------------------------------------- REP_SAYS
    A1 u A2 |- (M,Oi,Os) sat Q says f
```

Failure

Fails unless M, O_i, O_s, P, Q, and f match in all three theorems and have the same types.

Example

```
- val th1 =
   ACL_ASSUM ``(reps Alice Bob f):('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat reps Alice Bob f : thm
- val th2 =
   ACL_ASSUM
   ``((Alice quoting Bob) says f):('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat Alice quoting Bob says f : thm
- REP_SAYS th1 th2;
> val it =  [..] |- (M,Oi,Os) sat Bob says f : thm
```

Implementation

```
fun REP_SAYS th1 th2 =
     MATCH_MP (MATCH_MP (SPEC_ALL Rep_Says) th1) th2;
```

See also

`REPS`

A.43 SAYS

`SAYS`	`(acl_infRules)`

`SAYS : term -> thm -> thm`

Synopsis

Applies the Says inference rule to a theorem `A |- (M,Oi,Os) sat f` in the access-control logic.

Description

The `SAYS` rule applied to a principal P and a theorem of the form `A |- (M,Oi,Os) sat f` produces the theorem `A |- (M,Oi,Os) sat (P says f)`.

```
     A |- (M,Oi,Os) sat f
---------------------------- SAYS P
A |- (M,Oi,Os) sat P says f
```

Failure

Fails unless the types of f and P are consistent and P is of type `'pname Princ`.

Example

The following example shows how theorem `th1 = |- (M,Oi,Os) sat p orf notf p` is modified by ``SAYS ``Alice:'c Princ`` ``.

```
- val th1 = ACL_TAUT ``(p orf notf p):('a,'c,'d,'e)Form``;
> val th1 = |- (M,Oi,Os) sat p orf notf p : thm
- SAYS ``Alice:'c Princ`` th1;
> val it = |- (M,Oi,Os) sat Alice says (p orf notf p) : thm
```

Implementation

```
fun SAYS Q th = (SPEC Q (MATCH_MP Says th));
```

See also

MP_SAYS

A.44 SAYS_SIMP1

```
SAYS_SIMP1                    (acl_infRules)
```

`SAYS_SIMP1 : thm -> thm`

Synopsis

Applies the Says_Simplification1 rule to conjunctive statements within says statements in theorems in the access-control logic.

Description

```
A |- (M,Oi,Os) sat P says (f1 andf f2)
-------------------------------------- SAYS_SIMP1
       A |- (M,Oi,Os) sat P says f1
```

Failure
Fails unless the input theorem is a conjunction within a says statement in the access-control logic.

Example

```
- val th =
   ACL_ASSUM
   ``(Alice says (p andf q)):('propVar,'pName,'Int,'Sec)Form``;
> val th =  [.] |- (M,Oi,Os) sat Alice says (p andf q) : thm
- SAYS_SIMP1 th;
> val it =  [.] |- (M,Oi,Os) sat Alice says p : thm
```

Implementation

```
fun SAYS_SIMP1 th = MATCH_MP (SPEC_ALL Says_Simplification1) th;
```

See also
SAYS_SIMP2, ACL_SIMP1, ACL_SIMP2

A.45 SAYS_SIMP2

```
SAYS_SIMP2                        (acl_infRules)
```

SAYS_SIMP2 : thm -> thm

Synopsis
Applies the Says_Simplification2 rule to conjunctive statements within says statements in theorems in the access-control logic.

Description

```
A |- (M,Oi,Os) sat P says (f1 andf f2)
-------------------------------------- SAYS_SIMP2
       A |- (M,Oi,Os) sat P says f2
```

Failure

Fails unless the input theorem is a conjunction within a says statement in the access-control logic.

Example

```
- val th =
   ACL_ASSUM
   ``(Alice says (p andf q)):('propVar,'pName,'Int,'Sec)Form``;
> val th =  [.] |- (M,Oi,Os) sat Alice says (p andf q) : thm
- SAYS_SIMP2 th;
> val it =  [.] |- (M,Oi,Os) sat Alice says q : thm
```

Implementation

```
fun SAYS_SIMP2 th = MATCH_MP (SPEC_ALL Says_Simplification2) th;
```

See also

SAYS_SIMP1, ACL_SIMP1, ACL_SIMP2

A.46 SL_DOMS

```
SL_DOMS                          (acl_infRules)
```

SL_DOMS : thm -> thm -> thm -> thm

Synopsis

Applies *sl_doms* to theorems in the access-control logic.

Description

```
       A1 |- (M,Oi,Os) sat sl P eqs l1
       A2 |- (M,Oi,Os) sat sl Q eqs l2
       A3 |- (M,Oi,Os) sat l2 doms l1
 -------------------------------------------- SL_DOMS
 A1 u A2 u A3 |- (M,Oi,Os) sat sl Q doms sl P
```

Failure

Fails unless the types are consistent across the three input theorems.

Example

The following example shows that when `l2 doms l1`, `(sl Alice) eqs l1`, and `(sl Bob) eqs l2`, then `(sl Bob) doms (sl Alice)`, i.e., when l_2 doms l_1, Alice's and Bob's security levels are l_1 and l_2, respectively, then Bob's security level dominates Alice's.

```
- val th1 =
   ACL_ASSUM
   ``((sl Alice) eqs l1):('propVar,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat sl Alice eqs l1 : thm
- val th2 =
   ACL_ASSUM ``((sl Bob) eqs l2):('propVar,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat sl Bob eqs l2 : thm
- val levelsRelation =
   ACL_ASSUM ``(l2 doms l1):('propVar,'pName,'Int,'Sec)Form``;
> val levelsRelation =  [.] |- (M,Oi,Os) sat l2 doms l1 : thm
- SL_DOMS th1 th2 levelsRelation;
> val it =  [...] |- (M,Oi,Os) sat sl Bob doms sl Alice : thm
```

Implementation

```
fun SL_DOMS th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP sl_doms th1) th2) th3;
```

See also

`DOMI_TRANS, DOMS_TRANS, IL_DOMI`

A.47 SPEAKS_FOR

```
SPEAKS_FOR                    (acl_infRules)
```

`SPEAKS_FOR : thm -> thm -> thm`

Synopsis

Applies *Derived Speaks For* to theorems in the access-control logic.

Description

```
A1 |- (M,Oi,Os) sat P speaks_for Q
   A2 |- (M,Oi,Os) sat P says f
---------------------------------- SPEAKS_FOR
 A1 u A2 |- (M,Oi,Os) sat Q says f
```

Failure

Fails unless the first theorem is of the form `P speaksfor Q`, the second is `P says f`, and the types are the same.

Example

```
- val th1 =
   ACL_ASSUM
   ``(Alice speaks_for Bob):('propVar, 'pName, 'Int, 'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat Alice speaks_for Bob : thm
- val th2 =
   ACL_ASSUM ``(Alice says f):('propVar, 'pName, 'Int, 'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat Alice says f : thm
- SPEAKS_FOR th1 th2;
> val it =  [..] |- (M,Oi,Os) sat Bob says f : thm
```

Implementation

```
fun SPEAKS_FOR th1 th2 =
    MATCH_MP (MATCH_MP (SPEC_ALL Derived_Speaks_For) th1) th2;
```

See also

`TRANS_SPEAKS_FOR, IDEMP_SPEAKS_FOR, MONO_SPEAKS_FOR, SAYS`

A.48 TRANS_SPEAKS_FOR

```
TRANS_SPEAKS_FOR          (acl_infRules)
```

`TRANS_SPEAKS_FOR : thm -> thm -> thm`

Synopsis

Applies Trans_Speaks_For to theorems in the access-control logic.

Description

```
  A1 |- (M,Oi,Os) sat P speaks_for Q
  A2 |- (M,Oi,Os) sat Q speaks_for R
----------------------------------------- TRANS_SPEAKS_FOR
  A1 u A2 |- (M,Oi,Os) sat P speaks_for R
```

Failure

Fails unless the types are consistent among the two theorems.

Example

```
- val th1 = ACL_ASSUM ``(P speaks_for Q):('Prop,'pName,'Int,'Sec)Form``;
> val th1 =  [.] |- (M,Oi,Os) sat P speaks_for Q : thm
- val th2 = ACL_ASSUM ``(Q speaks_for R):('Prop,'pName,'Int,'Sec)Form``;
> val th2 =  [.] |- (M,Oi,Os) sat Q speaks_for R : thm
- TRANS_SPEAKS_FOR th1 th2;
> val it =  [..] |- (M,Oi,Os) sat P speaks_for R : thm
```

Implementation

```
fun TRANS_SPEAKS_FOR th1 th2 =
   (MATCH_MP (MATCH_MP Trans_Speaks_For th1) th2);
```

See also

SPEAKS_FOR, IDEMP_SPEAKS_FOR, MONO_SPEAKS_FOR, SAYS

Appendix B

Access-Control Logic Tactics in HOL

B.1 ACL_CONJ_TAC

ACL_CONJ_TAC	(acl_infRules)

```
ACL_CONJ_TAC : ('a * term) -> (('a * term) list * (thm list -> thm))
```

Synopsis
Reduces an ACL conjunctive goal to two separate subgoals.

Description
When applied to a goal `A ?- (M,Oi,Os) sat t1 andf t2`, reduces it to the two subgoals corresponding to each conjunct separately.

```
        A ?- (M,Oi,Os) sat t1 andf t2
 ===========================================  ACL_CONJ_TAC
 A ?- (M,Oi,Os) sat t1   A ?- (M,Oi,Os) sat t2
```

Failure
Fails unless the conclusion of the goal is an ACL conjunction.

Example
Applying `ACL_CONJ_TAC` to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p andf q
         ------------------------------------
           0.  (M,Oi,Os) sat q
           1.  (M,Oi,Os) sat p

     : proofs
```

produces the following subgoals:

```
2 subgoals:
> val it =

    (M,Oi,Os) sat q
    ------------------------------------
      0.  (M,Oi,Os) sat q
      1.  (M,Oi,Os) sat p


    (M,Oi,Os) sat p
    ------------------------------------
      0.  (M,Oi,Os) sat q
      1.  (M,Oi,Os) sat p

    2 subgoals
     : proof
```

Implementation

The implementation is as follows

```
fun ACL_CONJ_TAC (asl,term) =
let
  val (tuple,conj) = dest_sat term
  val (conj1,conj2) = dest_andf conj
  val conjTerm1 = mk_sat (tuple,conj1)
  val conjTerm2 = mk_sat (tuple,conj2)
in
 ([(asl,conjTerm1),(asl,conjTerm2)],
  fn [th1,th2] => ACL_CONJ th1 th2)
end
```

See also

B.2 ACL_DISJ1_TAC

```
ACL_DISJ1_TAC                (acl_infRules)
```

```
ACL_DISJ1_TAC : ('a * term) -> (('a * term) list * (thm list -> thm))
```

Synopsis

Selects the left disjunct of an ACL disjunctive goal.

Description

When applied to a goal `A ?- (M,Oi,Os) sat t1 orf t2`, the tactic `ACL_DISJ1_TAC` reduces it to the subgoal corresponding to the left disjunct.

```
     A ?- (M,Oi,Os) sat t1 orf t2
  ================================  ACL_DISJ1_TAC
       A ?- (M,Oi,Os) sat t1
```

Failure

Fails unless the goal is an ACL disjunction.

Example

Applying `ACL_DISJ1_TAC` to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p orf q
         ------------------------------------
           (M,Oi,Os) sat p

     : proofs
```

yields the following subgoal:

```
1 subgoal:
> val it =

    (M,Oi,Os) sat p
    ------------------------------------
      (M,Oi,Os) sat p
     : proof
```

Implementation

```
fun ACL_DISJ1_TAC (asl,term) =
let
  val (tuple,disj) = dest_sat term
  val (disj1,disj2) = dest_orf disj
  val disjTerm1 = mk_sat (tuple,disj1)
in
  ([(asl,disjTerm1)], fn [th] => ACL_DISJ1 disj2 th)
end
```

See also

ACL_DISJ2_TAC

B.3 ACL_DISJ2_TAC

ACL_DISJ2_TAC	(acl_infRules)

```
ACL_DISJ2_TAC : ('a * term) -> (('a * term) list * (thm list -> thm))
```

Synopsis

Selects the right disjunct of an ACL disjunctive goal.

Description

When applied to a goal `A ?- (M,Oi,Os) sat t1 orf t2`, the tactic `ACL_DISJ2_TAC` reduces it to the subgoal corresponding to the right disjunct.

```
      A ?- (M,Oi,Os) sat t1 orf t2
    =================================  ACL_DISJ2_TAC
        A ?- (M,Oi,Os) sat t2
```

Failure

Fails unless the goal is an ACL disjunction.

Example

Applying `ACL_DISJ2_TAC` to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p orf q
         ------------------------------------
           (M,Oi,Os) sat q

      : proofs
```

yields the following subgoal:

```
1 subgoal:
> val it =

    (M,Oi,Os) sat q
    ------------------------------------
      (M,Oi,Os) sat q
     : proof
```

Implementation

```
fun ACL_DISJ2_TAC (asl,term) =
let
  val (tuple,disj) = dest_sat term
  val (disj1,disj2) = dest_orf disj
  val disjTerm2 = mk_sat (tuple,disj2)
in
  ([(asl,disjTerm2)], fn [th] => ACL_DISJ2 disj1 th)
end
```

See also

ACL_DISJ1_TAC

B.4 ACL_MP_TAC

```
ACL_MP_TAC                    (acl_infRules)
```

```
ACL_MP_TAC : thm -> ('a * term) -> (('a * term) list * (thm list ->
    thm))
```

Synopsis

Reduces a goal to an ACL implication from a known theorem.

Description

When applied to the theorem `A' |- (M,Oi,Os) sat s` and the goal `A ?- (M,Oi,Os) sat t`, the tactic `ACL_MP_TAC` reduces the goal to `A ?- (M,Oi,Os) sat s impf t`. Unless A' is a subset of A, this is an invalid tactic.

```
          A ?- (M,Oi,Os) sat t
   =================================  ACL_MP_TAC (A' |- s)
       A ?- (M,Oi,Os) sat s impf t
```

Failure

Fails unless A' is a subset of A.

Example

Applying `ACL_MP_TAC` to the theorem `(M,Oi,Os) sat q |- (M,Oi,Os) sat q` and the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p
         ------------------------------------
           (M,Oi,Os) sat q
           (M,Oi,Os) sat q impf p

     : proofs
```

yields the following subgoal:

```
1 subgoal:
> val it =

    (M,Oi,Os) sat q impf p
    ------------------------------------
      (M,Oi,Os) sat q impf p
     : proof
```

Implementation

```
fun ACL_MP_TAC thb (asl,term) =
let
  val (tuple,form) = dest_sat term
  val (ntuple,nform) = dest_sat (concl thb)
  val newForm = mk_impf (nform,form)
  val newTerm = mk_sat (tuple,newForm)
  val predTerm = mk_sat (tuple,nform)
in
    ([(asl,newTerm)], fn [th] => ACL_MP thb th)
end
```

See also

B.5 ACL_AND_SAYS_RL_TAC

ACL_AND_SAYS_RL_TAC (acl_infRules)

```
ACL_AND_SAYS_RL_TAC : ('a * term) -> (('a * term) list * (thm list ->
    thm))
```

Synopsis

Rewrites a goal with *meet* to two *says* statements.

Description

When applied to a goal `A ?- (M,Oi,Os) sat p meet q says f`, returns a new subgoal in the form `A ?- (M,Oi,Os) sat (p says f) andf (q says f)`.

```
    A ?- (M,Oi,Os) sat p meet q says f
  =======================================  ACL_AND_SAYS_RL_TAC
   A ?- (M,Oi,Os) sat (p says f)
                           andf (q says f)
```

Failure

Fails unless the goal is in the form `p meet q says f`.

Example

Applying `ACL_AND_SAYS_RL_TAC` to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p meet q says f
         ------------------------------------
           (M,Oi,Os) sat p says f andf q says f

     : proofs
```

yields the following subgoal:

```
1 subgoal:
> val it =

    (M,Oi,Os) sat p says f andf q says f
    ------------------------------------
      (M,Oi,Os) sat p says f andf q says f
     : proof
```

Implementation

```
fun ACL_AND_SAYS_RL_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princs,prop) = dest_says form
 val (princ1,princ2) = dest_meet princs
 val conj1 = mk_says (princ1,prop)
 val conj2 = mk_says (princ2,prop)
 val conj = mk_andf (conj1,conj2)
 val newTerm = mk_sat (tuple,conj)
in
 ([(asl,newTerm)], fn [th] => AND_SAYS_RL th)
end
```

See also

ACL_AND_SAYS_LR_TAC

B.6 ACL_AND_SAYS_LR_TAC

```
ACL_AND_SAYS_LR_TAC   (acl_infRules)
```

ACL_AND_SAYS_LR_TAC : ('a * term) -> (('a * term) list * (thm list -> thm))

Synopsis

Rewrites a goal with conjunctive *says* statements into a *meet* statement.

Description

When applied to a goal `A ?- (M,Oi,Os) sat (p says f) andf (q says f)`, returns a new subgoal in the form `A ?- (M,Oi,Os) sat p meet q says f`.

```
  A ?- (M,Oi,Os) sat (p says f)
                      andf (q says f)
 ========================================  ACL_AND_SAYS_LR_TAC
   A ?- (M,Oi,Os) sat p meet q says f
```

Failure

Fails unless the goal is in the form `(p says f) andf (q says f)`.

Example

Applying `ACL_AND_SAYS_LR_TAC` to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p says f andf q says f
         ------------------------------------
           (M,Oi,Os) sat p meet q says f

     : proofs
```

yields the following subgoal:

```
1 subgoal:
> val it =

    (M,Oi,Os) sat p meet q says f
    ------------------------------------
      (M,Oi,Os) sat p meet q says f
     : proof
```

Implementation

```
fun ACL_AND_SAYS_LR_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (conj1,conj2) = dest_andf form
 val (princ1,prop) = dest_says conj1
 val (princ2,_) = dest_says conj2
 val princs = mk_meet (princ1,princ2)
 val newForm = mk_says (princs,prop)
 val newTerm = mk_sat (tuple,newForm)
in
 ([(asl,newTerm)], fn [th] => AND_SAYS_LR th)
end
```

See also

B.7 ACL_CONTROLS_TAC

ACL_CONTROLS_TAC (acl_infRules)

```
ACL_CONTROLS_TAC : term -> ('a * term) -> (('a * term) list * (thm
    list -> thm))
```

Synopsis
Reduces a goal to corresponding *controls* and *says* subgoals.

Description
When applied to a *princ* `p` and a goal `A ?- (M,Oi,Os) sat f`, returns a two new subgoals in the form `A ?- (M,Oi,Os) sat p controls f` and `A ?- (M,Oi,Os) sat p says f`.

```
         A ?- (M,Oi,Os) sat f
  =======================================  ACL_CONTROLS_TAC p
      A ?- (M,Oi,Os) sat p controls f
      A ?- (M,Oi,Os) sat p says f
```

Failure
Fails unless the goal is a form type and *p* is a principle.

Example
Applying `ACL_CONTROLS_TAC` to principle *p* and the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat f
         ------------------------------------
           0.  (M,Oi,Os) sat p says f
           1.  (M,Oi,Os) sat p controls f

     : proofs
```

yields the following subgoals:

```
2 subgoals:
> val it =

    (M,Oi,Os) sat p says f
    ------------------------------------
      0.  (M,Oi,Os) sat p says f
      1.  (M,Oi,Os) sat p controls f


    (M,Oi,Os) sat p controls f
    ------------------------------------
      0.  (M,Oi,Os) sat p says f
      1.  (M,Oi,Os) sat p controls f

    2 subgoals
     : proof
```

Implementation

```
fun ACL_CONTROLS_TAC princ (asl,term) =
let
 val (tuple,form) = dest_sat term
 val newControls = mk_controls (princ,form)
 val newTerm1 = mk_sat (tuple,newControls)
 val newSays = mk_says (princ,form)
 val newTerm2 = mk_sat (tuple,newSays)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2] => CONTROLS th1 th2)
end
```

See also

B.8 ACL_DC_TAC

ACL_DC_TAC	(acl_infRules)

```
ACL_DC_TAC : term -> ('a * term) -> (('a * term) list * (thm list ->
    thm))
```

Synopsis

Reduces a goal to corresponding *controls* and *speaks_for* subgoals.

Description

When applied to a principal `q` and a goal `A ?- (M,Oi,Os) sat p controls f`, returns a two new subgoals in the form `A ?- (M,Oi,Os) sat p speaks_for q` and `A ?- (M,Oi,Os) sat q controls f`.

```
      A ?- (M,Oi,Os) sat p controls f
  =======================================  ACL_DC_TAC q
     A ?- (M,Oi,Os) sat p speaks_for q
     A ?- (M,Oi,Os) sat q controls  f
```

Failure

Fails unless the goal is an ACL controls statement and *q* is a principle.

Example

Applying `ACL_DC_TAC` to principal *q* and the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p controls f
         ------------------------------------
           0.  (M,Oi,Os) sat q controls f
           1.  (M,Oi,Os) sat p speaks_for q

     : proofs
```

yields the following subgoals:

```
2 subgoals:
> val it =

    (M,Oi,Os) sat q controls f
    ------------------------------------
      0.  (M,Oi,Os) sat q controls f
      1.  (M,Oi,Os) sat p speaks_for q


    (M,Oi,Os) sat p speaks_for q
    ------------------------------------
      0.  (M,Oi,Os) sat q controls f
      1.  (M,Oi,Os) sat p speaks_for q

    2 subgoals
     : proof
```

Implementation

```
fun ACL_DC_TAC princ2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ1,prop) = dest_controls form
 val formType = type_of form
 val speaksFor = ``(^princ1 speaks_for ^princ2)
                                  :^(ty_antiq formType)``
 val newTerm1 = mk_sat (tuple,speaksFor)
 val newControls = mk_controls (princ2,prop)
 val newTerm2 = mk_sat (tuple,newControls)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2] => DC th1 th2)
end
```

See also

B.9 ACL_DOMI_TRANS_TAC

```
ACL_DOMI_TRANS_TAC     (acl_infRules)
```

```
ACL_DOMI_TRANS_TAC : term -> ('a * term) -> (('a * term) list * (thm
    list -> thm))
```

Synopsis

Reduces a goal to two subgoals using the transitive property of integrity levels.

Description

When applied to an integrity level `l2` and a goal `A ?- (M,Oi,Os) sat l1 domi l3`, returns a two new subgoals in the form `A ?- (M,Oi,Os) sat l1 domi l2` and `A ?- (M,Oi,Os) sat l2 domi l3`.

```
     A ?- (M,Oi,Os) sat l1 domi l3
 =======================================  ACL_DOMI_TRANS_TAC l2
     A ?- (M,Oi,Os) sat l1 domi l2
     A ?- (M,Oi,Os) sat l2 domi l3
```

Failure

Fails unless the goal is an ACL domi statement and `l2` is an integrity level.

Example

Applying `ACL_DOMI_TRANS_TAC` to integrity level `l2` and the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat l1 domi l3
         ------------------------------------
           0.  (M,Oi,Os) sat l2 domi l3
           1.  (M,Oi,Os) sat l1 domi l2

     : proofs
```

yields the following subgoals:

```
2 subgoals:
> val it =

    (M,Oi,Os) sat l2 domi l3
    ------------------------------------
      0.  (M,Oi,Os) sat l2 domi l3
      1.  (M,Oi,Os) sat l1 domi l2


    (M,Oi,Os) sat l1 domi l2
    ------------------------------------
      0.  (M,Oi,Os) sat l2 domi l3
      1.  (M,Oi,Os) sat l1 domi l2

    2 subgoals
     : proof
```

Implementation

```
fun ACL_DOMI_TRANS_TAC iLev2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (iLev1,iLev3) = dest_domi form
 val formType = type_of form
 val newDomi1 = ``(^iLev1 domi ^iLev2):^(ty_antiq formType)``
 val newTerm1 = mk_sat (tuple,newDomi1)
 val newDomi2 = ``(^iLev2 domi ^iLev3):^(ty_antiq formType)``
 val newTerm2 = mk_sat (tuple,newDomi2)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2]
                                        => DOMI_TRANS th1 th2)
end
```

See also

B.10 ACL_DOMS_TRANS_TAC

```
ACL_DOMS_TRANS_TAC    (acl_infRules)
```

ACL_DOMS_TRANS_TAC : term -> ('a * term) -> (('a * term) list * (thm list -> thm))

Synopsis

Reduces a goal to two subgoals using the transitive property of security levels.

Description

When applied to a security level `l2` and a goal `A ?- (M,Oi,Os) sat l1 doms l3`, returns a two new subgoals in the form `A ?- (M,Oi,Os) sat l1 doms l2` and `A ?- (M,Oi,Os) sat l2 doms l3`.

```
      A ?- (M,Oi,Os) sat l1 doms l3
  ========================================  ACL_DOMS_TRANS_TAC l2
      A ?- (M,Oi,Os) sat l1 doms l2
      A ?- (M,Oi,Os) sat l2 doms l3
```

Failure

Fails unless the goal is an ACL doms statement and `l2` is a security level.

Example

Applying `ACL_DOMS_TRANS_TAC` to security level `l2` and the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat l1 doms l3
         ------------------------------------
           0.  (M,Oi,Os) sat l2 doms l3
           1.  (M,Oi,Os) sat l1 doms l2

     : proofs
```

yields the following subgoals:

```
2 subgoals:
> val it =

    (M,Oi,Os) sat l2 doms l3
    ------------------------------------
      0.  (M,Oi,Os) sat l2 doms l3
      1.  (M,Oi,Os) sat l1 doms l2


    (M,Oi,Os) sat l1 doms l2
    ------------------------------------
      0.  (M,Oi,Os) sat l2 doms l3
      1.  (M,Oi,Os) sat l1 doms l2

    2 subgoals
     : proof
```

Implementation

```
fun ACL_DOMS_TRANS_TAC sLev2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (sLev1,sLev3) = dest_doms form
 val formType = type_of form
 val newDoms1 = ``(^sLev1 doms ^sLev2):^(ty_antiq formType)``
 val newTerm1 = mk_sat (tuple,newDoms1)
 val newDoms2 = ``(^sLev2 doms ^sLev3):^(ty_antiq formType)``
 val newTerm2 = mk_sat (tuple,newDoms2)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2]
                                    => DOMS_TRANS th1 th2)
end
```

See also

B.11 ACL_HS_TAC

ACL_HS_TAC	(acl_infRules)

```
ACL_HS_TAC : term -> ('a * term) -> (('a * term) list * (thm list ->
   thm))
```

Synopsis

Reduces a goal to two subgoals using the transitive property of ACL implications.

Description

When applied to an ACL formula `f2` and a goal `A ?- (M,Oi,Os) sat f1 impf f3`, returns a two new subgoals in the form `A ?- (M,Oi,Os) sat f1 impf f2` and `A ?- (M,Oi,Os) sat f2 impf f3`.

```
       A ?- (M,Oi,Os) sat f1 impf f3
  ========================================  ACL_HS_TAC f2
      A ?- (M,Oi,Os) sat f1 impf f2
      A ?- (M,Oi,Os) sat f2 impf f3
```

Failure

Fails unless the goal is an ACL implication and `f2` is an ACL formula.

Example

Applying `ACL_HS_TAC` to ACL formula `f2` and the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat f1 impf f3
         ------------------------------------
           0.  (M,Oi,Os) sat f2 impf f3
           1.  (M,Oi,Os) sat f1 impf f2

     : proofs
```

yields the following subgoals:

```
2 subgoals:
> val it =

    (M,Oi,Os) sat f2 impf f3
    ------------------------------------
      0.  (M,Oi,Os) sat f2 impf f3
      1.  (M,Oi,Os) sat f1 impf f2


    (M,Oi,Os) sat f1 impf f2
    ------------------------------------
      0.  (M,Oi,Os) sat f2 impf f3
      1.  (M,Oi,Os) sat f1 impf f2

    2 subgoals
     : proof
```

Implementation

```
fun ACL_HS_TAC f2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (f1,f3) = dest_impf form
 val newImpf1 = mk_impf (f1,f2)
 val newTerm1 = mk_sat (tuple,newImpf1)
 val newImpf2 = mk_impf (f2,f3)
 val newTerm2 = mk_sat (tuple,newImpf2)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2] => HS th1 th2)
end
```

See also

B.12 ACL_IDEMP_SPEAKS_FOR_TAC

```
ACL_IDEMP_SPEAKS_FOR_TAC                                    (acl_infRules)
```

```
ACL_IDEMP_SPEAKS_FOR_TAC : ('a * term) -> (('a * term) list * (thm
    list -> thm))
```

Synopsis
Proves a goal of the form p speaks_for p.

Description
When applied to a goal `A ?- (M,Oi,Os) sat p speaks_for p`, it will prove the goal.

```
   A ?- (M,Oi,Os) sat p speaks_for p
 ===================================  ACL_IDEMP_SPEAKS_FOR_TAC
```

Failure
Fails unless the goal is an ACL formula of the form p speaks_for p.

Example
Applying `ACL_IDEMP_SPEAKS_FOR_TAC` to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p speaks_for p


     : proofs
```

yields the following result:

```
    Initial goal proved.
    |- (M,Oi,Os) sat p speaks_for p
     : proof
```

Implementation

```
fun ACL_IDEMP_SPEAKS_FOR_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ1,princ2) = dest_speaks_for form
 val th1 = IDEMP_SPEAKS_FOR princ1
 val tupleType = type_of tuple
 val (_,[kripketype,_]) = dest_type tupleType
 val (_,[_,btype,_,_,_]) = dest_type kripketype
 val formType = type_of form
 val (_,[proptype,princtype,inttype,sectype]) = dest_type formType
 val th2 = INST_TYPE [``:'a`` |-> proptype, ``:'b`` |-> btype,
                      ``:'d`` |-> inttype, ``:'e`` |-> sectype] th1
in
 ([], fn xs => th2)
end
```

See also

B.13 ACL_IL_DOMI_TAC

ACL_IL_DOMI_TAC	(acl_infRules)

ACL_IL_DOMI_TAC : term -> ('a * term) -> (('a * term) list * (thm list -> thm))

Synopsis

Reduces a goal comparing integrity levels of two principals to three subgoals.

Description

When applied to a goal `A ?- (M,Oi,Os) sat il q domi il p`, integrity levels `l2` and `l1` it will return 3 subgoals.

```
      A ?- (M,Oi,Os) sat il q domi il p
 ========================================  ACL_IL_DOMI_TAC
     A ?- (M,Oi,Os) sat l2 domi l1
     A ?- (M,Oi,Os) sat il q eqi l2
     A ?- (M,Oi,Os) sat il p eqi l1
```

Failure

Fails unless the goal is an ACL formula of the form il q domi il p.

Example

Applying `ACL_IDEMP_SPEAKS_FOR_TAC` to integrity levels `l2`, `l1` and the following goal:

```
     1. Incomplete goalstack:
          Initial goal:

          (M,Oi,Os) sat il q domi il p
          ------------------------------------
            0.  (M,Oi,Os) sat l2 domi l1
            1.  (M,Oi,Os) sat il q eqi l2
            2.  (M,Oi,Os) sat il p eqi l1

      : proofs
```

yields the following three subgoals:

```
3 subgoals:
> val it =

     (M,Oi,Os) sat l2 domi l1
     ------------------------------------
       0.  (M,Oi,Os) sat l2 domi l1
       1.  (M,Oi,Os) sat il q eqi l2
       2.  (M,Oi,Os) sat il p eqi l1


     (M,Oi,Os) sat il p eqi l1
     ------------------------------------
       0.  (M,Oi,Os) sat l2 domi l1
       1.  (M,Oi,Os) sat il q eqi l2
       2.  (M,Oi,Os) sat il p eqi l1


     (M,Oi,Os) sat il q eqi l2
     ------------------------------------
       0.  (M,Oi,Os) sat l2 domi l1
       1.  (M,Oi,Os) sat il q eqi l2
       2.  (M,Oi,Os) sat il p eqi l1

     3 subgoals
      : proof
```

Implementation

```
fun ACL_IL_DOMI_TAC ilev1 ilev2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val formtype = type_of form
 val (ilevprinc1,ilevprinc2) = dest_domi form
 val princ1eq = ``(^ilevprinc1 eqi ^ilev1):^(ty_antiq formtype)``
 val subgoal1 = mk_sat (tuple,princ1eq)
 val princ2eq = ``(^ilevprinc2 eqi ^ilev2):^(ty_antiq formtype)``
 val subgoal2 = mk_sat (tuple,princ2eq)
 val ilevdomi = ``(^ilev1 domi ^ilev2):^(ty_antiq formtype)``
 val subgoal3 = mk_sat (tuple,ilevdomi)
in
 ([(asl,subgoal1),(asl,subgoal2),(asl,subgoal3)],
   fn [th1,th2,th3] => IL_DOMI th2 th1 th3)
end
```

See also

B.14 ACL_MONO_SPEAKS_FOR_TAC

```
ACL_MONO_SPEAKS_FOR_TAC                                (acl_infRules)
```

```
ACL_MONO_SPEAKS_FOR_TAC : term -> ('a * term) -> (('a * term) list *
   (thm list -> thm))
```

Synopsis

Reduces a goal to corresponding *speaks_for* subgoals.

Description

When applied to a goal `A ?- (M,Oi,Os) sat (p quoting q) speaks_for (p' quoting q')`, it will return 2 subgoals.

```
A ?- (M,Oi,Os) sat (p quoting q) speaks_for (p' quoting q')
============================================================  ACL_MONO_
         A ?- (M,Oi,Os) sat p speaks_for p'
         A ?- (M,Oi,Os) sat q speaks_for q'
```

Failure

Fails unless the goal is an ACL formula of the form (p quoting q) speaks_for (p' quoting q').

Example

Applying ACL_MONO_SPEAKS_FOR_TAC to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p quoting q speaks_for p' quoting q'
         ------------------------------------
           0.  (M,Oi,Os) sat q speaks_for q'
           1.  (M,Oi,Os) sat p speaks_for p'

     : proofs
```

yields the following two subgoals:

```
2 subgoals:
> val it =

    (M,Oi,Os) sat q speaks_for q'
    ------------------------------------
      0.  (M,Oi,Os) sat q speaks_for q'
      1.  (M,Oi,Os) sat p speaks_for p'


    (M,Oi,Os) sat p speaks_for p'
    ------------------------------------
      0.  (M,Oi,Os) sat q speaks_for q'
      1.  (M,Oi,Os) sat p speaks_for p'

    2 subgoals
     : proof
```

Implementation

```
fun ACL_MONO_SPEAKS_FOR_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val formtype = type_of form
 val (quote1,quote2) = dest_speaks_for form
 val (princ1,princ2) = dest_quoting quote1
 val (princ1',princ2') = dest_quoting quote2
 val speaksfor1 = ``(^princ1 speaks_for ^princ1'):^(ty_antiq formtype)``
 val subgoal1 = mk_sat (tuple,speaksfor1)
 val speaksfor2 = ``(^princ2 speaks_for ^princ2'):^(ty_antiq formtype)``
 val subgoal2 = mk_sat (tuple,speaksfor2)
in
 ([(asl,subgoal1),(asl,subgoal2)], fn [th1,th2] => MONO_SPEAKS_FOR th1 th2)
end
```

See also

B.15 ACL_MP_SAYS_TAC

ACL_MP_SAYS_TAC	(acl_infRules)

`ACL_MP_SAYS_TAC : term -> ('a * term) -> (('a * term) list * (thm list -> thm))`

Synopsis

Proves a goal of the form `A ?- (M,Oi,Os) sat (p says (f1 impf f2)) impf ((p says f1) impf (p says f2))`

Description

When applied to a goal `A ?- (M,Oi,Os) sat (p says (f1 impf f2)) impf ((p says f1) impf (p says f2))`, it will prove the goal.

```
A ?- (M,Oi,Os) sat
      (p says (f1 impf f2)) impf
       ((p says f1) impf (p says f2))
======================================= ACL_MP_SAYS_TAC
```

Failure

Fails unless the goal is an ACL formula of the form (p says (f1 impf f2)) impf ((p says f1) impf (p says f2)).

Example

Applying `ACL_MP_SAYS_TAC` to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p says (f1 impf f2) impf p says f1 impf p says f2


     : proofs
```

yields the following result:

```
    Initial goal proved.
    |- (M,Oi,Os) sat p says (f1 impf f2) impf p says f1 impf p says f2
     : proof
```

Implementation

```
fun ACL_MP_SAYS_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (saysterm,_) = dest_impf form
 val (princ,impterm) = dest_says saysterm
 val (f1,f2) = dest_impf impterm
  val tupleType = type_of tuple
 val (_,[kripketype,_]) = dest_type tupleType
 val (_,[_,btype,_,_,_]) = dest_type kripketype
 val th1 = MP_SAYS princ f1 f2
 val th2 = INST_TYPE [``:'b`` |-> btype] th1
in
 ([], fn xs => th2)
end
```

See also

B.16 ACL_QUOTING_LR_TAC

```
ACL_QUOTING_LR_TAC    (acl_infRules)
```

`ACL_QUOTING_LR_TAC : term -> ('a * term) -> (('a * term) list * (thm list -> thm))`

Synopsis
Reduces a *says* goal to corresponding *quoting* subgoal.

Description
When applied to a goal `A ?- (M,Oi,Os) sat p says q says f`, it will return 1 subgoal.

```
   A ?- (M,Oi,Os) sat p says q says f
  ======================================= ACL_QUOTING_LR_TAC
   A ?- (M,Oi,Os) sat p quoting q says f
```

Failure

Fails unless the goal is an ACL formula of the form p says q says f.

Example

Applying ACL_QUOTING_LR_TAC to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p says q says f
         ------------------------------------
           (M,Oi,Os) sat p quoting q says f

     : proofs
```

yields the following subgoal:

```
1 subgoal:
> val it =

    (M,Oi,Os) sat p quoting q says f
    ------------------------------------
      (M,Oi,Os) sat p quoting q says f
     : proof
```

Implementation

```
fun ACL_QUOTING_LR_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ1,saysterm) = dest_says form
 val (princ2,f) = dest_says saysterm
 val quotingterm = mk_quoting (princ1,princ2)
 val newform = mk_says (quotingterm,f)
 val subgoal = mk_sat (tuple,newform)
in
 ([(asl,subgoal)], fn [th] => QUOTING_LR th)
end
```

See also

B.17 ACL_QUOTING_RL_TAC

```
ACL_QUOTING_RL_TAC    (acl_infRules)
```

`ACL_QUOTING_RL_TAC : term -> ('a * term) -> (('a * term) list * (thm list -> thm))`

Synopsis
Reduces a *quoting* goal to corresponding *says* subgoal.

Description
When applied to a goal `A ?- (M,Oi,Os) sat p quoting q says f`, it will return 1 subgoal.

```
  A ?- (M,Oi,Os) sat p quoting q says f
======================================= ACL_QUOTING_RL_TAC
  A ?- (M,Oi,Os) sat p says q says f
```

Failure
Fails unless the goal is an ACL formula of the form p quoting q says f.

Example
Applying `ACL_QUOTING_RL_TAC` to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p quoting q says f
         ------------------------------------
           (M,Oi,Os) sat p says q says f

     : proofs
```

yields the following subgoal:

```
1 subgoal:
> val it =

    (M,Oi,Os) sat p says q says f
    ------------------------------------
      (M,Oi,Os) sat p says q says f
     : proof
```

Implementation

```
fun ACL_QUOTING_RL_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (quotingterm,f) = dest_says form
 val (princ1,princ2) = dest_quoting quotingterm
 val saysterm = mk_says (princ2,f)
 val newform = mk_says (princ1,saysterm)
 val subgoal = mk_sat (tuple,newform)
in
 ([(asl,subgoal)], fn [th] => QUOTING_RL th)
end
```

See also

B.18 ACL_REPS_TAC

ACL_REPS_TAC	(acl_infRules)

`ACL_REPS_TAC : term -> ('a * term) -> (('a * term) list * (thm list -> thm))`

Synopsis

Reduces a goal to the corresponding *reps* subgoals.

Description

When applied to principals `p`, `q` and a goal `A ?- (M,Oi,Os) sat f`, it will return 3 subgoals.

```
         A ?- (M,Oi,Os) sat f
  ======================================== ACL_REPS_TAC
      A ?- (M,Oi,Os) sat q controls f
    A ?- (M,Oi,Os) sat p quoting q says f
       A ?- (M,Oi,Os) sat reps p q f
```

Failure

Fails unless the goal is an ACL formula.

Example

Applying `ACL_REPS_TAC` to principals `p`, `q` and the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat f
         ------------------------------------
           0.  (M,Oi,Os) sat q controls f
           1.  (M,Oi,Os) sat p quoting q says f
           2.  (M,Oi,Os) sat reps p q f

     : proofs
```

yields the following 3 subgoals:

```
3 subgoals:
> val it =

    (M,Oi,Os) sat q controls f
    ------------------------------------
      0.  (M,Oi,Os) sat q controls f
      1.  (M,Oi,Os) sat p quoting q says f
      2.  (M,Oi,Os) sat reps p q f


    (M,Oi,Os) sat p quoting q says f
    ------------------------------------
      0.  (M,Oi,Os) sat q controls f
      1.  (M,Oi,Os) sat p quoting q says f
      2.  (M,Oi,Os) sat reps p q f


    (M,Oi,Os) sat reps p q f
    ------------------------------------
      0.  (M,Oi,Os) sat q controls f
      1.  (M,Oi,Os) sat p quoting q says f
      2.  (M,Oi,Os) sat reps p q f

    3 subgoals
     : proof
```

Implementation

```
fun ACL_REPS_TAC princ1 princ2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val repterm = mk_reps (princ1,princ2,form)
 val subgoal1 = mk_sat (tuple,repterm)
 val quotingterm = mk_quoting (princ1,princ2)
 val saysterm = mk_says (quotingterm,form)
 val subgoal2 = mk_sat (tuple,saysterm)
 val controlsterm = mk_controls (princ2,form)
 val subgoal3 = mk_sat (tuple,controlsterm)
in
 ([(asl,subgoal1),(asl,subgoal2),(asl,subgoal3)], fn [th1,th2,th3] => REPS th1
end
```

See also

B.19 ACL_REP_SAYS_TAC

```
ACL_REP_SAYS_TAC        (acl_infRules)
```

```
ACL_REP_SAYS_TAC : term -> ('a * term) -> (('a * term) list * (thm list -> thm)
```

Synopsis

Reduces a *says* goal to the corresponding *reps* subgoals.

Description

When applied to principal `p` and a goal `A ?- (M,Oi,Os) sat q says f`, it will return 2 subgoals.

```
         A ?- (M,Oi,Os) sat q says f
  ======================================== ACL_REP_SAYS_TAC
    A ?- (M,Oi,Os) sat p quoting q says f
       A ?- (M,Oi,Os) sat reps p q f
```

Failure

Fails unless the goal is an ACL formula in the form of q says f.

Example

Applying `ACL_REP_SAYS_TAC` to principal `p` and the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat q says f
         ------------------------------------
           0.  (M,Oi,Os) sat p quoting q says f
           1.  (M,Oi,Os) sat reps p q f

     : proofs
```

yields the following 2 subgoals:

```
2 subgoals:
> val it =

    (M,Oi,Os) sat p quoting q says f
    ------------------------------------
      0.  (M,Oi,Os) sat p quoting q says f
      1.  (M,Oi,Os) sat reps p q f


    (M,Oi,Os) sat reps p q f
    ------------------------------------
      0.  (M,Oi,Os) sat p quoting q says f
      1.  (M,Oi,Os) sat reps p q f

    2 subgoals
     : proof
```

Implementation

```
fun ACL_REP_SAYS_TAC princ1 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ2,f) = dest_says form
 val repsterm = mk_reps (princ1,princ2,f)
 val subgoal1 = mk_sat (tuple,repsterm)
 val quotingterm = mk_quoting (princ1,princ2)
 val saysterm = mk_says (quotingterm,f)
 val subgoal2 = mk_sat (tuple,saysterm)
in
 ([(asl,subgoal1),(asl,subgoal2)], fn [th1,th2] => REP_SAYS th1 th2)
end
```

See also

B.20 ACL_SAYS_TAC

```
ACL_SAYS_TAC                    (acl_infRules)
```

`ACL_SAYS_TAC : term -> ('a * term) -> (('a * term) list * (thm list -> thm))`

Synopsis

Reduces a *says* goal to the corresponding subgoal.

Description

When applied to principal a goal `A ?- (M,Oi,Os) sat p says f`, it will return 1 subgoal.

```
    A ?- (M,Oi,Os) sat p says f
  =============================== ACL_SAYS_TAC
       A ?- (M,Oi,Os) sat f
```

Failure

Fails unless the goal is an ACL formula in the form of p says f.

Example

Applying `ACL_SAYS_TAC` to the following goal:

```
    1. Incomplete goalstack:
         Initial goal:

         (M,Oi,Os) sat p says f
         ------------------------------------
           (M,Oi,Os) sat f

     : proofs
```

yields the following subgoal:

```
1 subgoal:
> val it =

    (M,Oi,Os) sat f
    ------------------------------------
      (M,Oi,Os) sat f
     : proof
```

Implementation

```
fun ACL_SAYS_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ,f) = dest_says form
 val subgoal = mk_sat (tuple,f)
in
 ([(asl,subgoal)], fn [th] => SAYS princ th)
end
```

See also

Appendix C

Access-Control Logic Theories in HOL

C.1 aclfoundation Theory

Built: 19 January 2017

Parent Theories: list

C.1.1 Datatypes

```
Form =
    TT
  | FF
  | prop 'aavar
  | notf (('aavar, 'apn, 'il, 'sl) Form)
  | (andf) (('aavar, 'apn, 'il, 'sl) Form)
           (('aavar, 'apn, 'il, 'sl) Form)
  | (orf) (('aavar, 'apn, 'il, 'sl) Form)
          (('aavar, 'apn, 'il, 'sl) Form)
  | (impf) (('aavar, 'apn, 'il, 'sl) Form)
           (('aavar, 'apn, 'il, 'sl) Form)
  | (eqf) (('aavar, 'apn, 'il, 'sl) Form)
          (('aavar, 'apn, 'il, 'sl) Form)
  | (says) ('apn Princ) (('aavar, 'apn, 'il, 'sl) Form)
  | (speaks_for) ('apn Princ) ('apn Princ)
  | (controls) ('apn Princ) (('aavar, 'apn, 'il, 'sl) Form)
  | reps ('apn Princ) ('apn Princ)
         (('aavar, 'apn, 'il, 'sl) Form)
  | (domi) (('apn, 'il) IntLevel) (('apn, 'il) IntLevel)
  | (eqi) (('apn, 'il) IntLevel) (('apn, 'il) IntLevel)
  | (doms) (('apn, 'sl) SecLevel) (('apn, 'sl) SecLevel)
  | (eqs) (('apn, 'sl) SecLevel) (('apn, 'sl) SecLevel)
  | (eqn) num num
  | (lte) num num
  | (lt) num num

Kripke =
    KS ('aavar -> 'aaworld -> bool)
       ('apn -> 'aaworld -> 'aaworld -> bool) ('apn -> 'il)
       ('apn -> 'sl)
```

```
Princ =
    Name 'apn
  | (meet) ('apn Princ) ('apn Princ)
  | (quoting) ('apn Princ) ('apn Princ) ;

IntLevel = iLab 'il | il 'apn ;

SecLevel = sLab 'sl | sl 'apn
```

C.1.2 Definitions

[imapKS_def]

```
⊢ ∀Intp Jfn ilmap slmap.
    imapKS (KS Intp Jfn ilmap slmap) = ilmap
```

[intpKS_def]

```
⊢ ∀Intp Jfn ilmap slmap.
    intpKS (KS Intp Jfn ilmap slmap) = Intp
```

[jKS_def]

```
⊢ ∀Intp Jfn ilmap slmap. jKS (KS Intp Jfn ilmap slmap) = Jfn
```

[O1_def]

```
⊢ O1 = PO one_weakorder
```

[one_weakorder_def]

```
⊢ ∀x y. one_weakorder x y ⟺ T
```

[po_TY_DEF]

```
⊢ ∃rep. TYPE_DEFINITION WeakOrder rep
```

[po_tybij]

```
⊢ (∀a. PO (repPO a) = a) ∧
  ∀r. WeakOrder r ⟺ (repPO (PO r) = r)
```

[prod_PO_def]

```
⊢ ∀PO1 PO2.
    prod_PO PO1 PO2 = PO (RPROD (repPO PO1) (repPO PO2))
```

[smapKS_def]

```
⊢ ∀Intp Jfn ilmap slmap.
    smapKS (KS Intp Jfn ilmap slmap) = slmap
```

[Subset_PO_def]

```
⊢ Subset_PO = PO (⊆)
```

C.1.3 Theorems

[abs_po11]

$\vdash \forall r\ r'.$
$\quad \texttt{WeakOrder}\ r \Rightarrow \texttt{WeakOrder}\ r' \Rightarrow ((\texttt{PO}\ r = \texttt{PO}\ r') \iff (r = r'))$

[absPO_fn_onto]

$\vdash \forall a.\ \exists r.\ (a = \texttt{PO}\ r) \wedge \texttt{WeakOrder}\ r$

[antisym_prod_antisym]

$\vdash \forall r\ s.$
$\quad \texttt{antisymmetric}\ r \wedge \texttt{antisymmetric}\ s \Rightarrow$
$\quad \texttt{antisymmetric}\ (\texttt{RPROD}\ r\ s)$

[EQ_WeakOrder]

$\vdash \texttt{WeakOrder}\ (=)$

[KS_bij]

$\vdash \forall M.\ M = \texttt{KS}\ (\texttt{intpKS}\ M)\ (\texttt{jKS}\ M)\ (\texttt{imapKS}\ M)\ (\texttt{smapKS}\ M)$

[one_weakorder_WO]

$\vdash \texttt{WeakOrder one\_weakorder}$

[onto_po]

$\vdash \forall r.\ \texttt{WeakOrder}\ r \iff \exists a.\ r = \texttt{repPO}\ a$

[po_bij]

$\vdash (\forall a.\ \texttt{PO}\ (\texttt{repPO}\ a) = a) \wedge$
$\quad \forall r.\ \texttt{WeakOrder}\ r \iff (\texttt{repPO}\ (\texttt{PO}\ r) = r)$

[PO_repPO]

$\vdash \forall a.\ \texttt{PO}\ (\texttt{repPO}\ a) = a$

[refl_prod_refl]

$\vdash \forall r\ s.\ \texttt{reflexive}\ r \wedge \texttt{reflexive}\ s \Rightarrow \texttt{reflexive}\ (\texttt{RPROD}\ r\ s)$

[repPO_iPO_partial_order]

$\vdash (\forall x.\ \texttt{repPO}\ iPO\ x\ x) \wedge$
$\quad (\forall x\ y.\ \texttt{repPO}\ iPO\ x\ y \wedge \texttt{repPO}\ iPO\ y\ x \Rightarrow (x = y)) \wedge$
$\quad \forall x\ y\ z.\ \texttt{repPO}\ iPO\ x\ y \wedge \texttt{repPO}\ iPO\ y\ z \Rightarrow \texttt{repPO}\ iPO\ x\ z$

[repPO_O1]

$\vdash \texttt{repPO O1 = one\_weakorder}$

[repPO_prod_PO]

$\vdash \forall po_1\ po_2.$
  repPO (prod_PO po_1 po_2) = RPROD (repPO po_1) (repPO po_2)

[repPO_Subset_PO]

$\vdash$ repPO Subset_PO = ($\subseteq$)

[RPROD_THM]

$\vdash \forall r\ s\ a\ b.$
  RPROD r s a b $\iff$ r (FST a) (FST b) $\land$ s (SND a) (SND b)

[SUBSET_WO]

$\vdash$ WeakOrder ($\subseteq$)

[trans_prod_trans]

$\vdash \forall r\ s.$ transitive r $\land$ transitive s $\Rightarrow$ transitive (RPROD r s)

[WeakOrder_Exists]

$\vdash \exists R.$ WeakOrder R

[WO_prod_WO]

$\vdash \forall r\ s.$ WeakOrder r $\land$ WeakOrder s $\Rightarrow$ WeakOrder (RPROD r s)

[WO_repPO]

$\vdash \forall r.$ WeakOrder r $\iff$ (repPO (PO r) = r)

C.2 aclsemantics Theory

Built: 19 January 2017

Parent Theories: aclfoundation

C.2.1 Definitions

[Efn_def]

$\vdash$ ($\forall Oi\ Os\ M.$ Efn Oi Os M TT = $\mathcal{U}$(:'v)) $\land$
  ($\forall Oi\ Os\ M.$ Efn Oi Os M FF = {}) $\land$
  ($\forall Oi\ Os\ M\ p.$ Efn Oi Os M (prop p) = intpKS M p) $\land$
  ($\forall Oi\ Os\ M\ f.$
    Efn Oi Os M (notf f) = $\mathcal{U}$(:'v) DIFF Efn Oi Os M f) $\land$
  ($\forall Oi\ Os\ M\ f_1\ f_2.$
    Efn Oi Os M (f_1 andf f_2) =
    Efn Oi Os M f_1 $\cap$ Efn Oi Os M f_2) $\land$

```
(∀Oi Os M f1 f2.
   Efn Oi Os M (f1 orf f2) =
   Efn Oi Os M f1 ∪ Efn Oi Os M f2) ∧
(∀Oi Os M f1 f2.
   Efn Oi Os M (f1 impf f2) =
   𝒰(:'v) DIFF Efn Oi Os M f1 ∪ Efn Oi Os M f2) ∧
(∀Oi Os M f1 f2.
   Efn Oi Os M (f1 eqf f2) =
   (𝒰(:'v) DIFF Efn Oi Os M f1 ∪ Efn Oi Os M f2) ∩
   (𝒰(:'v) DIFF Efn Oi Os M f2 ∪ Efn Oi Os M f1)) ∧
(∀Oi Os M P f.
   Efn Oi Os M (P says f) =
   {w | Jext (jKS M) P w ⊆ Efn Oi Os M f}) ∧
(∀Oi Os M P Q.
   Efn Oi Os M (P speaks_for Q) =
   if Jext (jKS M) Q RSUBSET Jext (jKS M) P then 𝒰(:'v)
   else {}) ∧
(∀Oi Os M P f.
   Efn Oi Os M (P controls f) =
   𝒰(:'v) DIFF {w | Jext (jKS M) P w ⊆ Efn Oi Os M f} ∪
   Efn Oi Os M f) ∧
(∀Oi Os M P Q f.
   Efn Oi Os M (reps P Q f) =
   𝒰(:'v) DIFF
   {w | Jext (jKS M) (P quoting Q) w ⊆ Efn Oi Os M f} ∪
   {w | Jext (jKS M) Q w ⊆ Efn Oi Os M f}) ∧
(∀Oi Os M intl1 intl2.
   Efn Oi Os M (intl1 domi intl2) =
   if repPO Oi (Lifn M intl2) (Lifn M intl1) then 𝒰(:'v)
   else {}) ∧
(∀Oi Os M intl2 intl1.
   Efn Oi Os M (intl2 eqi intl1) =
   (if repPO Oi (Lifn M intl2) (Lifn M intl1) then 𝒰(:'v)
    else {}) ∩
   if repPO Oi (Lifn M intl1) (Lifn M intl2) then 𝒰(:'v)
   else {}) ∧
(∀Oi Os M secl1 secl2.
   Efn Oi Os M (secl1 doms secl2) =
   if repPO Os (Lsfn M secl2) (Lsfn M secl1) then 𝒰(:'v)
   else {}) ∧
(∀Oi Os M secl2 secl1.
   Efn Oi Os M (secl2 eqs secl1) =
   (if repPO Os (Lsfn M secl2) (Lsfn M secl1) then 𝒰(:'v)
    else {}) ∩
   if repPO Os (Lsfn M secl1) (Lsfn M secl2) then 𝒰(:'v)
   else {}) ∧
(∀Oi Os M numExp1 numExp2.
```

```
    Efn Oi Os M (numExp₁ eqn numExp₂) =
    if numExp₁ = numExp₂ then 𝒰(:'v) else {}) ∧
  (∀Oi Os M numExp₁ numExp₂.
    Efn Oi Os M (numExp₁ lte numExp₂) =
    if numExp₁ ≤ numExp₂ then 𝒰(:'v) else {}) ∧
  ∀Oi Os M numExp₁ numExp₂.
    Efn Oi Os M (numExp₁ lt numExp₂) =
    if numExp₁ < numExp₂ then 𝒰(:'v) else {}
```

[Jext_def]

```
⊢ (∀J s. Jext J (Name s) = J s) ∧
  (∀J P₁ P₂.
     Jext J (P₁ meet P₂) = Jext J P₁ RUNION Jext J P₂) ∧
  ∀J P₁ P₂. Jext J (P₁ quoting P₂) = Jext J P₂ O Jext J P₁
```

[Lifn_def]

```
⊢ (∀M l. Lifn M (iLab l) = l) ∧
  ∀M name. Lifn M (il name) = imapKS M name
```

[Lsfn_def]

```
⊢ (∀M l. Lsfn M (sLab l) = l) ∧
  ∀M name. Lsfn M (sl name) = smapKS M name
```

C.2.2 Theorems

[andf_def]

```
⊢ ∀Oi Os M f₁ f₂.
    Efn Oi Os M (f₁ andf f₂) = Efn Oi Os M f₁ ∩ Efn Oi Os M f₂
```

[controls_def]

```
⊢ ∀Oi Os M P f.
    Efn Oi Os M (P controls f) =
    𝒰(:'v) DIFF {w | Jext (jKS M) P w ⊆ Efn Oi Os M f} ∪
    Efn Oi Os M f
```

[controls_says]

```
⊢ ∀M P f.
    Efn Oi Os M (P controls f) = Efn Oi Os M (P says f impf f)
```

[domi_def]

```
⊢ ∀Oi Os M intl₁ intl₂.
    Efn Oi Os M (intl₁ domi intl₂) =
    if repPO Oi (Lifn M intl₂) (Lifn M intl₁) then 𝒰(:'v)
    else {}
```

[doms_def]

$\vdash \forall Oi\ Os\ M\ secl_1\ secl_2.$
`Efn` $Oi\ Os\ M$ `(`$secl_1$ `doms` $secl_2$`) =`
if `repPO` Os `(Lsfn` $M\ secl_2$`) (Lsfn` $M\ secl_1$`)` **then** $\mathcal{U}$`(:'v)`
else $\{\}$

[eqf_def]

$\vdash \forall Oi\ Os\ M\ f_1\ f_2.$
`Efn` $Oi\ Os\ M$ `(`f_1 `eqf` f_2`) =`
`(`$\mathcal{U}$`(:'v) DIFF Efn` $Oi\ Os\ M\ f_1$ $\cup$ `Efn` $Oi\ Os\ M\ f_2$`)` $\cap$
`(`$\mathcal{U}$`(:'v) DIFF Efn` $Oi\ Os\ M\ f_2$ $\cup$ `Efn` $Oi\ Os\ M\ f_1$`)`

[eqf_impf]

$\vdash \forall M\ f_1\ f_2.$
`Efn` $Oi\ Os\ M$ `(`f_1 `eqf` f_2`) =`
`Efn` $Oi\ Os\ M$ `((`f_1 `impf` f_2`) andf (`f_2 `impf` f_1`))`

[eqi_def]

$\vdash \forall Oi\ Os\ M\ intl_2\ intl_1.$
`Efn` $Oi\ Os\ M$ `(`$intl_2$ `eqi` $intl_1$`) =`
`(`**if** `repPO` Oi `(Lifn` $M\ intl_2$`) (Lifn` $M\ intl_1$`)` **then** $\mathcal{U}$`(:'v)`
else $\{\}$`)` $\cap$
if `repPO` Oi `(Lifn` $M\ intl_1$`) (Lifn` $M\ intl_2$`)` **then** $\mathcal{U}$`(:'v)`
else $\{\}$

[eqi_domi]

$\vdash \forall M\ intL_1\ intL_2.$
`Efn` $Oi\ Os\ M$ `(`$intL_1$ `eqi` $intL_2$`) =`
`Efn` $Oi\ Os\ M$ `(`$intL_2$ `domi` $intL_1$ `andf` $intL_1$ `domi` $intL_2$`)`

[eqn_def]

$\vdash \forall Oi\ Os\ M\ numExp_1\ numExp_2.$
`Efn` $Oi\ Os\ M$ `(`$numExp_1$ `eqn` $numExp_2$`) =`
if $numExp_1 = numExp_2$ **then** $\mathcal{U}$`(:'v)` **else** $\{\}$

[eqs_def]

$\vdash \forall Oi\ Os\ M\ secl_2\ secl_1.$
`Efn` $Oi\ Os\ M$ `(`$secl_2$ `eqs` $secl_1$`) =`
`(`**if** `repPO` Os `(Lsfn` $M\ secl_2$`) (Lsfn` $M\ secl_1$`)` **then** $\mathcal{U}$`(:'v)`
else $\{\}$`)` $\cap$
if `repPO` Os `(Lsfn` $M\ secl_1$`) (Lsfn` $M\ secl_2$`)` **then** $\mathcal{U}$`(:'v)`
else $\{\}$

[eqs_doms]

$\vdash \forall M\ secL_1\ secL_2.$
$\quad \texttt{Efn}\ Oi\ Os\ M\ (secL_1\ \texttt{eqs}\ secL_2) =$
$\quad \texttt{Efn}\ Oi\ Os\ M\ (secL_2\ \texttt{doms}\ secL_1\ \texttt{andf}\ secL_1\ \texttt{doms}\ secL_2)$

[FF_def]

$\vdash \forall Oi\ Os\ M.\ \texttt{Efn}\ Oi\ Os\ M\ \texttt{FF} = \{\}$

[impf_def]

$\vdash \forall Oi\ Os\ M\ f_1\ f_2.$
$\quad \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{impf}\ f_2) =$
$\quad \mathcal{U}(:'\texttt{v})\ \texttt{DIFF}\ \texttt{Efn}\ Oi\ Os\ M\ f_1 \cup \texttt{Efn}\ Oi\ Os\ M\ f_2$

[lt_def]

$\vdash \forall Oi\ Os\ M\ numExp_1\ numExp_2.$
$\quad \texttt{Efn}\ Oi\ Os\ M\ (numExp_1\ \texttt{lt}\ numExp_2) =$
$\quad \textbf{if}\ numExp_1 < numExp_2\ \textbf{then}\ \mathcal{U}(:'\texttt{v})\ \textbf{else}\ \{\}$

[lte_def]

$\vdash \forall Oi\ Os\ M\ numExp_1\ numExp_2.$
$\quad \texttt{Efn}\ Oi\ Os\ M\ (numExp_1\ \texttt{lte}\ numExp_2) =$
$\quad \textbf{if}\ numExp_1 \leq numExp_2\ \textbf{then}\ \mathcal{U}(:'\texttt{v})\ \textbf{else}\ \{\}$

[meet_def]

$\vdash \forall J\ P_1\ P_2.\ \texttt{Jext}\ J\ (P_1\ \texttt{meet}\ P_2) = \texttt{Jext}\ J\ P_1\ \texttt{RUNION}\ \texttt{Jext}\ J\ P_2$

[name_def]

$\vdash \forall J\ s.\ \texttt{Jext}\ J\ (\texttt{Name}\ s) = J\ s$

[notf_def]

$\vdash \forall Oi\ Os\ M\ f.\ \texttt{Efn}\ Oi\ Os\ M\ (\texttt{notf}\ f) = \mathcal{U}(:'\texttt{v})\ \texttt{DIFF}\ \texttt{Efn}\ Oi\ Os\ M\ f$

[orf_def]

$\vdash \forall Oi\ Os\ M\ f_1\ f_2.$
$\quad \texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{orf}\ f_2) = \texttt{Efn}\ Oi\ Os\ M\ f_1 \cup \texttt{Efn}\ Oi\ Os\ M\ f_2$

[prop_def]

$\vdash \forall Oi\ Os\ M\ p.\ \texttt{Efn}\ Oi\ Os\ M\ (\texttt{prop}\ p) = \texttt{intpKS}\ M\ p$

[quoting_def]

$\vdash \forall J\ P_1\ P_2.\ \texttt{Jext}\ J\ (P_1\ \texttt{quoting}\ P_2) = \texttt{Jext}\ J\ P_2\ \texttt{O}\ \texttt{Jext}\ J\ P_1$

[reps_def]

$\vdash \forall Oi\ Os\ M\ P\ Q\ f.$
  Efn Oi Os M (reps P Q f) =
  $\mathcal{U}$(:'v) DIFF
  $\{w$ | Jext (jKS M) (P quoting Q) $w \subseteq$ Efn Oi Os M $f\} \cup$
  $\{w$ | Jext (jKS M) Q $w \subseteq$ Efn Oi Os M $f\}$

[says_def]

$\vdash \forall Oi\ Os\ M\ P\ f.$
  Efn Oi Os M (P says f) =
  $\{w$ | Jext (jKS M) P $w \subseteq$ Efn Oi Os M $f\}$

[speaks_for_def]

$\vdash \forall Oi\ Os\ M\ P\ Q.$
  Efn Oi Os M (P speaks_for Q) =
  if Jext (jKS M) Q RSUBSET Jext (jKS M) P **then** $\mathcal{U}$(:'v)
  else { }

[TT_def]

$\vdash \forall Oi\ Os\ M.$ Efn Oi Os M TT = $\mathcal{U}$(:'v)

C.3 aclrules Theory

Built: 19 January 2017

Parent Theories: aclsemantics

C.3.1 Definitions

[sat_def]

$\vdash \forall M\ Oi\ Os\ f.$ (M,Oi,Os) sat f $\iff$ (Efn Oi Os M f = $\mathcal{U}$(:'world))

C.3.2 Theorems

[And_Says]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
  (M,Oi,Os) sat P meet Q says f eqf P says f andf Q says f

[And_Says_Eq]

$\vdash$ (M,Oi,Os) sat P meet Q says f $\iff$
  (M,Oi,Os) sat P says f andf Q says f

[and_says_lemma]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{meet}\ Q\ \texttt{says}\ f\ \texttt{impf}\ P\ \texttt{says}\ f\ \texttt{andf}\ Q\ \texttt{says}\ f$

[Controls_Eq]

$\vdash \forall M\ Oi\ Os\ P\ f.$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{controls}\ f \iff (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{says}\ f\ \texttt{impf}\ f$

[DIFF_UNIV_SUBSET]

$\vdash (\mathcal{U}(:'a)\ \texttt{DIFF}\ s \cup t = \mathcal{U}(:'a)) \iff s \subseteq t$

[domi_antisymmetric]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2.$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{domi}\ l_2 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_2\ \texttt{domi}\ l_1 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{eqi}\ l_2$

[domi_reflexive]

$\vdash \forall M\ Oi\ Os\ l.\ (M,Oi,Os)\ \texttt{sat}\ l\ \texttt{domi}\ l$

[domi_transitive]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2\ l_3.$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{domi}\ l_2 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_2\ \texttt{domi}\ l_3 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{domi}\ l_3$

[doms_antisymmetric]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2.$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{doms}\ l_2 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_2\ \texttt{doms}\ l_1 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{eqs}\ l_2$

[doms_reflexive]

$\vdash \forall M\ Oi\ Os\ l.\ (M,Oi,Os)\ \texttt{sat}\ l\ \texttt{doms}\ l$

[doms_transitive]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2\ l_3.$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{doms}\ l_2 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_2\ \texttt{doms}\ l_3 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ l_1\ \texttt{doms}\ l_3$

[eqf_and_impf]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
$\quad (M,Oi,Os)\ \texttt{sat}\ f_1\ \texttt{eqf}\ f_2 \iff$
$\quad (M,Oi,Os)\ \texttt{sat}\ (f_1\ \texttt{impf}\ f_2)\ \texttt{andf}\ (f_2\ \texttt{impf}\ f_1)$

[eqf_andf1]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{andf}\ g \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f'\ \texttt{andf}\ g$

[eqf_andf2]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ g\ \texttt{andf}\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ g\ \texttt{andf}\ f'$

[eqf_controls]

$\vdash \forall M\ Oi\ Os\ P\ f\ f'.$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{controls}\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ P\ \texttt{controls}\ f'$

[eqf_eq]

$\vdash (\texttt{Efn}\ Oi\ Os\ M\ (f_1\ \texttt{eqf}\ f_2) = \mathcal{U}(\texttt{:'b})) \iff$
$\quad (\texttt{Efn}\ Oi\ Os\ M\ f_1 = \texttt{Efn}\ Oi\ Os\ M\ f_2)$

[eqf_eqf1]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{eqf}\ g \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f'\ \texttt{eqf}\ g$

[eqf_eqf2]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ g\ \texttt{eqf}\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ g\ \texttt{eqf}\ f'$

[eqf_impf1]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{impf}\ g \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ f'\ \texttt{impf}\ g$

[eqf_impf2]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
$\quad (M, Oi, Os)\ \texttt{sat}\ f\ \texttt{eqf}\ f' \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ g\ \texttt{impf}\ f \Rightarrow$
$\quad (M, Oi, Os)\ \texttt{sat}\ g\ \texttt{impf}\ f'$

[eqf_notf]

$\vdash \forall M\ Oi\ Os\ f\ f'.$
(M,Oi,Os) sat f eqf $f' \Rightarrow$
(M,Oi,Os) sat notf $f \Rightarrow$
(M,Oi,Os) sat notf f'

[eqf_orf1]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
(M,Oi,Os) sat f eqf $f' \Rightarrow$
(M,Oi,Os) sat f orf $g \Rightarrow$
(M,Oi,Os) sat f' orf g

[eqf_orf2]

$\vdash \forall M\ Oi\ Os\ f\ f'\ g.$
(M,Oi,Os) sat f eqf $f' \Rightarrow$
(M,Oi,Os) sat g orf $f \Rightarrow$
(M,Oi,Os) sat g orf f'

[eqf_reps]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f\ f'.$
(M,Oi,Os) sat f eqf $f' \Rightarrow$
(M,Oi,Os) sat reps $P\ Q\ f \Rightarrow$
(M,Oi,Os) sat reps $P\ Q\ f'$

[eqf_sat]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
(M,Oi,Os) sat f_1 eqf $f_2 \Rightarrow$
$((M,Oi,Os)$ sat $f_1 \iff (M,Oi,Os)$ sat $f_2)$

[eqf_says]

$\vdash \forall M\ Oi\ Os\ P\ f\ f'.$
(M,Oi,Os) sat f eqf $f' \Rightarrow$
(M,Oi,Os) sat P says $f \Rightarrow$
(M,Oi,Os) sat P says f'

[eqi_Eq]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2.$
(M,Oi,Os) sat l_1 eqi $l_2 \iff$
(M,Oi,Os) sat l_2 domi l_1 andf l_1 domi l_2

[eqs_Eq]

$\vdash \forall M\ Oi\ Os\ l_1\ l_2.$
(M,Oi,Os) sat l_1 eqs $l_2 \iff$
(M,Oi,Os) sat l_2 doms l_1 andf l_1 doms l_2

[Idemp_Speaks_For]

$\vdash \forall M\ Oi\ Os\ P.\ (M,Oi,Os)$ sat P speaks_for P

[Image_cmp]

$\vdash \forall R_1\ R_2\ R_3\ u.\ (R_1 \circ R_2)\ u \subseteq R_3 \iff R_2\ u \subseteq \{y \mid R_1\ y \subseteq R_3\}$

[Image_SUBSET]

$\vdash \forall R_1\ R_2.\ R_2$ RSUBSET $R_1 \Rightarrow \forall w.\ R_2\ w \subseteq R_1\ w$

[Image_UNION]

$\vdash \forall R_1\ R_2\ w.\ (R_1$ RUNION $R_2)\ w = R_1\ w \cup R_2\ w$

[INTER_EQ_UNIV]

$\vdash (s \cap t = \mathcal{U}(:'a)) \iff (s = \mathcal{U}(:'a)) \land (t = \mathcal{U}(:'a))$

[Modus_Ponens]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
(M,Oi,Os) sat $f_1 \Rightarrow$
(M,Oi,Os) sat f_1 impf $f_2 \Rightarrow$
(M,Oi,Os) sat f_2

[Mono_speaks_for]

$\vdash \forall M\ Oi\ Os\ P\ P'\ Q\ Q'.$
(M,Oi,Os) sat P speaks_for $P' \Rightarrow$
(M,Oi,Os) sat Q speaks_for $Q' \Rightarrow$
(M,Oi,Os) sat P quoting Q speaks_for P' quoting Q'

[MP_Says]

$\vdash \forall M\ Oi\ Os\ P\ f_1\ f_2.$
(M,Oi,Os) sat
P says $(f_1$ impf $f_2)$ impf P says f_1 impf P says f_2

[Quoting]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M,Oi,Os) sat P quoting Q says f eqf P says Q says f

[Quoting_Eq]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
(M,Oi,Os) sat P quoting Q says $f \iff$
(M,Oi,Os) sat P says Q says f

[reps_def_lemma]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
Efn $Oi\ Os\ M$ (reps $P\ Q\ f$) =
Efn $Oi\ Os\ M$ (P quoting Q says f impf Q says f)

[Reps_Eq]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M,Oi,Os)\ \texttt{sat}\ \texttt{reps}\ P\ Q\ f \iff$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{quoting}\ Q\ \texttt{says}\ f\ \texttt{impf}\ Q\ \texttt{says}\ f$

[sat_allworld]

$\vdash \forall M\ f.\ (M,Oi,Os)\ \texttt{sat}\ f \iff \forall w.\ w \in \texttt{Efn}\ Oi\ Os\ M\ f$

[sat_andf_eq_and_sat]

$\vdash (M,Oi,Os)\ \texttt{sat}\ f_1\ \texttt{andf}\ f_2 \iff$
$(M,Oi,Os)\ \texttt{sat}\ f_1 \wedge (M,Oi,Os)\ \texttt{sat}\ f_2$

[sat_TT]

$\vdash (M,Oi,Os)\ \texttt{sat}\ \texttt{TT}$

[Says]

$\vdash \forall M\ Oi\ Os\ P\ f.\ (M,Oi,Os)\ \texttt{sat}\ f \Rightarrow (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{says}\ f$

[says_and_lemma]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{says}\ f\ \texttt{andf}\ Q\ \texttt{says}\ f\ \texttt{impf}\ P\ \texttt{meet}\ Q\ \texttt{says}\ f$

[Speaks_For]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ Q\ \texttt{impf}\ P\ \texttt{says}\ f\ \texttt{impf}\ Q\ \texttt{says}\ f$

[speaks_for_SUBSET]

$\vdash \forall R_3\ R_2\ R_1.$
$\quad R_2\ \texttt{RSUBSET}\ R_1 \Rightarrow \forall w.\ \{w \mid R_1\ w \subseteq R_3\} \subseteq \{w \mid R_2\ w \subseteq R_3\}$

[SUBSET_Image_SUBSET]

$\vdash \forall R_1\ R_2\ R_3.$
$\quad (\forall w_1.\ R_2\ w_1 \subseteq R_1\ w_1) \Rightarrow$
$\quad \forall w.\ \{w \mid R_1\ w \subseteq R_3\} \subseteq \{w \mid R_2\ w \subseteq R_3\}$

[Trans_Speaks_For]

$\vdash \forall M\ Oi\ Os\ P\ Q\ R.$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ Q \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ Q\ \texttt{speaks\_for}\ R \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ R$

[UNIV_DIFF_SUBSET]

$\vdash \forall R_1\ R_2.\ R_1 \subseteq R_2 \Rightarrow (\mathcal{U}(:'\texttt{a})\ \texttt{DIFF}\ R_1 \cup R_2 = \mathcal{U}(:'\texttt{a}))$

[world_and]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2\ w.$
$\quad w \in \mathtt{Efn}\ Oi\ Os\ M\ (f_1\ \mathtt{andf}\ f_2) \iff$
$\quad w \in \mathtt{Efn}\ Oi\ Os\ M\ f_1 \land w \in \mathtt{Efn}\ Oi\ Os\ M\ f_2$

[world_eq]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2\ w.$
$\quad w \in \mathtt{Efn}\ Oi\ Os\ M\ (f_1\ \mathtt{eqf}\ f_2) \iff$
$\quad (w \in \mathtt{Efn}\ Oi\ Os\ M\ f_1 \iff w \in \mathtt{Efn}\ Oi\ Os\ M\ f_2)$

[world_eqn]

$\vdash \forall M\ Oi\ Os\ n_1\ n_2\ w.\ w \in \mathtt{Efn}\ Oi\ Os\ m\ (n_1\ \mathtt{eqn}\ n_2) \iff (n_1 = n_2)$

[world_F]

$\vdash \forall M\ Oi\ Os\ w.\ w \notin \mathtt{Efn}\ Oi\ Os\ M\ \mathtt{FF}$

[world_imp]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2\ w.$
$\quad w \in \mathtt{Efn}\ Oi\ Os\ M\ (f_1\ \mathtt{impf}\ f_2) \iff$
$\quad w \in \mathtt{Efn}\ Oi\ Os\ M\ f_1 \Rightarrow w \in \mathtt{Efn}\ Oi\ Os\ M\ f_2$

[world_lt]

$\vdash \forall M\ Oi\ Os\ n_1\ n_2\ w.\ w \in \mathtt{Efn}\ Oi\ Os\ m\ (n_1\ \mathtt{lt}\ n_2) \iff n_1 < n_2$

[world_lte]

$\vdash \forall M\ Oi\ Os\ n_1\ n_2\ w.\ w \in \mathtt{Efn}\ Oi\ Os\ m\ (n_1\ \mathtt{lte}\ n_2) \iff n_1 \leq n_2$

[world_not]

$\vdash \forall M\ Oi\ Os\ f\ w.\ w \in \mathtt{Efn}\ Oi\ Os\ M\ (\mathtt{notf}\ f) \iff w \notin \mathtt{Efn}\ Oi\ Os\ M\ f$

[world_or]

$\vdash \forall M\ f_1\ f_2\ w.$
$\quad w \in \mathtt{Efn}\ Oi\ Os\ M\ (f_1\ \mathtt{orf}\ f_2) \iff$
$\quad w \in \mathtt{Efn}\ Oi\ Os\ M\ f_1 \lor w \in \mathtt{Efn}\ Oi\ Os\ M\ f_2$

[world_says]

$\vdash \forall M\ Oi\ Os\ P\ f\ w.$
$\quad w \in \mathtt{Efn}\ Oi\ Os\ M\ (P\ \mathtt{says}\ f) \iff$
$\quad \forall v.\ v \in \mathtt{Jext}\ (\mathtt{jKS}\ M)\ P\ w \Rightarrow v \in \mathtt{Efn}\ Oi\ Os\ M\ f$

[world_T]

$\vdash \forall M\ Oi\ Os\ w.\ w \in \mathtt{Efn}\ Oi\ Os\ M\ \mathtt{TT}$

C.4 aclDrules Theory

Built: 19 January 2017

Parent Theories: aclrules

C.4.1 Theorems

[Conjunction]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
$\quad (M,Oi,Os)\ \texttt{sat}\ f_1 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ f_2 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ f_1\ \texttt{andf}\ f_2$

[Controls]

$\vdash \forall M\ Oi\ Os\ P\ f.$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{says}\ f \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{controls}\ f \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ f$

[Derived_Controls]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ Q \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ Q\ \texttt{controls}\ f \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{controls}\ f$

[Derived_Speaks_For]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{speaks\_for}\ Q \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ P\ \texttt{says}\ f \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ Q\ \texttt{says}\ f$

[Disjunction1]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.\ (M,Oi,Os)\ \texttt{sat}\ f_1 \Rightarrow (M,Oi,Os)\ \texttt{sat}\ f_1\ \texttt{orf}\ f_2$

[Disjunction2]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.\ (M,Oi,Os)\ \texttt{sat}\ f_2 \Rightarrow (M,Oi,Os)\ \texttt{sat}\ f_1\ \texttt{orf}\ f_2$

[Disjunctive_Syllogism]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
$\quad (M,Oi,Os)\ \texttt{sat}\ f_1\ \texttt{orf}\ f_2 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ \texttt{notf}\ f_1 \Rightarrow$
$\quad (M,Oi,Os)\ \texttt{sat}\ f_2$

[Double_Negation]

$\vdash \forall M\ Oi\ Os\ f.\ (M,Oi,Os)$ sat notf (notf f) $\Rightarrow$ (M,Oi,Os) sat f

[eqn_eqn]

$\vdash$ (M,Oi,Os) sat c_1 eqn n_1 $\Rightarrow$
(M,Oi,Os) sat c_2 eqn n_2 $\Rightarrow$
(M,Oi,Os) sat n_1 eqn n_2 $\Rightarrow$
(M,Oi,Os) sat c_1 eqn c_2

[eqn_lt]

$\vdash$ (M,Oi,Os) sat c_1 eqn n_1 $\Rightarrow$
(M,Oi,Os) sat c_2 eqn n_2 $\Rightarrow$
(M,Oi,Os) sat n_1 lt n_2 $\Rightarrow$
(M,Oi,Os) sat c_1 lt c_2

[eqn_lte]

$\vdash$ (M,Oi,Os) sat c_1 eqn n_1 $\Rightarrow$
(M,Oi,Os) sat c_2 eqn n_2 $\Rightarrow$
(M,Oi,Os) sat n_1 lte n_2 $\Rightarrow$
(M,Oi,Os) sat c_1 lte c_2

[Hypothetical_Syllogism]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2\ f_3.$
(M,Oi,Os) sat f_1 impf f_2 $\Rightarrow$
(M,Oi,Os) sat f_2 impf f_3 $\Rightarrow$
(M,Oi,Os) sat f_1 impf f_3

[il_domi]

$\vdash \forall M\ Oi\ Os\ P\ Q\ l_1\ l_2.$
(M,Oi,Os) sat il P eqi l_1 $\Rightarrow$
(M,Oi,Os) sat il Q eqi l_2 $\Rightarrow$
(M,Oi,Os) sat l_2 domi l_1 $\Rightarrow$
(M,Oi,Os) sat il Q domi il P

[INTER_EQ_UNIV]

$\vdash \forall s_1\ s_2.\ (s_1 \cap s_2 = \mathcal{U}(:'a)) \iff (s_1 = \mathcal{U}(:'a)) \land (s_2 = \mathcal{U}(:'a))$

[Modus_Tollens]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$
(M,Oi,Os) sat f_1 impf f_2 $\Rightarrow$
(M,Oi,Os) sat notf f_2 $\Rightarrow$
(M,Oi,Os) sat notf f_1

[Rep_Controls_Eq]

$\vdash \forall M\ Oi\ Os\ A\ B\ f.$
$\quad (M,Oi,Os)$ sat reps A B f $\iff$
$\quad (M,Oi,Os)$ sat A controls B says f

[Rep_Says]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M,Oi,Os)$ sat reps P Q f $\Rightarrow$
$\quad (M,Oi,Os)$ sat P quoting Q says f $\Rightarrow$
$\quad (M,Oi,Os)$ sat Q says f

[Reps]

$\vdash \forall M\ Oi\ Os\ P\ Q\ f.$
$\quad (M,Oi,Os)$ sat reps P Q f $\Rightarrow$
$\quad (M,Oi,Os)$ sat P quoting Q says f $\Rightarrow$
$\quad (M,Oi,Os)$ sat Q controls f $\Rightarrow$
$\quad (M,Oi,Os)$ sat f

[Says_Simplification1]

$\vdash \forall M\ Oi\ Os\ P\ f_1\ f_2.$
$\quad (M,Oi,Os)$ sat P says $(f_1$ andf $f_2)$ $\Rightarrow$ (M,Oi,Os) sat P says f_1

[Says_Simplification2]

$\vdash \forall M\ Oi\ Os\ P\ f_1\ f_2.$
$\quad (M,Oi,Os)$ sat P says $(f_1$ andf $f_2)$ $\Rightarrow$ (M,Oi,Os) sat P says f_2

[Simplification1]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$ (M,Oi,Os) sat f_1 andf f_2 $\Rightarrow$ (M,Oi,Os) sat f_1

[Simplification2]

$\vdash \forall M\ Oi\ Os\ f_1\ f_2.$ (M,Oi,Os) sat f_1 andf f_2 $\Rightarrow$ (M,Oi,Os) sat f_2

[sl_doms]

$\vdash \forall M\ Oi\ Os\ P\ Q\ l_1\ l_2.$
$\quad (M,Oi,Os)$ sat sl P eqs l_1 $\Rightarrow$
$\quad (M,Oi,Os)$ sat sl Q eqs l_2 $\Rightarrow$
$\quad (M,Oi,Os)$ sat l_2 doms l_1 $\Rightarrow$
$\quad (M,Oi,Os)$ sat sl Q doms sl P

Appendix D

Access-Control Logic Source Files

D.1 aclfoundation Theory

```
(* a.c.l. foundation, with arithmetic expressions by skc 2/11/09 *)

(* a.c.l. foundation, mutilated by lm, now with curried J. 1/25/09 *)
(* now with eqp removed (see Access/Plotkin) 1/29/09 *)

(** ACCESS CONTROL LOGIC FOUNDATION in our textbook *)
(***********************
* ACCESS CONTROL LOGIC FOUNDATION in our textbook
* The semantics of the logic is mainly built using Kripke structures.
* Sets Li, Ls of integrity and security labels (or levels) are
* considered part of the syntax, and their partial orders Oi, Os are
* separate parameters of the semantics and of the derivation system.
* The components of the
* Kripke structure <W, I, J, imap, smap>
* we use are:
* (1) W, a non-empty set of worlds,
* (2) I, an interpretation function mapping primitive propositions to the
*      sets of worlds where the propositions are true,
* (3) J, a function mapping principal expressions to relations on worlds,
* (4) imap, a function mapping each simple principal name
*      to an integrity level, and
* (5) smap, a function mapping each simple principal name
*      to a security level.
* *********************)

(**********************
* HOL IMPLEMENTATION APPROACH
* We introduce a Hol type Kripke that is parameterized on
* a non-empty set of worlds of arbitrary type, a set of integrity
* levels of arbitrary type, and a set of security levels of arbitrary
* type. At the end of this theory, we define Kripke as follows:
* val _ = Hol_datatype
*    `Kripke = KS of ('var -> ('world set)) =>
*                     ('pn -> ('world -> ('world set))) =>
*                     ('pn -> 'il) =>
*                     ('pn -> 'sl)`
* where:
* (1) 'world is a type variable for the type of possible worlds (note
*      that every type in Hol is non-empty),
* (2) 'var is a type variable for propositional variables,
* (3) 'pn a type variable for the type of simple principal names,
* (4) 'il is a type variable for the type of integrity levels, and
* (5) 'sl is a type variable for the type of security levels,
*
* To accomplish the above we do the following in order:
* (1) define the type of partial orders, po
* (2) define the type of principal expressions, Princ
* (3) define the type of integrity level expressions, IntLevel,
* (4) define the type of security level expressions, SecLevel,
* (5) define the type of formulas, Form, and
* (6) define the type of Kripke structures, Kripke.
```

```
************************)

(***********************
* THE DEFINITIONS START HERE
***********************)
(*
load "pred_setTheory";
load "relationTheory";
load "PairedLambda";
*)

structure aclfoundationScript =
struct

open HolKernel boolLib Parse;
open bossLib pred_setTheory relationTheory PairedLambda pairTheory oneTheory;

val _ = new_theory "aclfoundation";
(*******************
* DEFINE PARTIAL ORDER TYPE
* The "dominates" relation on security labels is a partial order (called a
* weak order in HOL - WeakOrder in relationTheory).  These relations are
* reflexive, antisymmetric, and transitive. What we want is a new type
* ('a)po, i.e., a type consisting of partial orderings on 'a.
* We now obtain this as follows: ************************************)

(* (1) Use predicate WeakOrder to select partial orderings from relations
       of type 'a.
   (2) Prove that the new type is non-empty - this we do by showing $= is
       a partial order (EQ_WeakOrder), then existentially quantifying $=,
       resulting in WeakOrder_Exists. *)

(* Show that $= satisfies WeakOrder, i.e., is a partial order *)
val EQ_WeakOrder =
    store_thm("EQ_WeakOrder",
        Term `WeakOrder ($=)`,
        REWRITE_TAC
        (map (SPEC ``($=):('a->'a->bool)``)
        [(INST_TYPE [Type`:'g` |-> Type `:'a`] WeakOrder),
          reflexive_def, antisymmetric_def, transitive_def]) THEN
        PROVE_TAC []);

(* Show that partial orders exist *)
val WeakOrder_Exists =
    save_thm
    ("WeakOrder_Exists",
     (EXISTS (Term `?R.WeakOrder R`, Term `$=`) EQ_WeakOrder));

(* WeakOrder_Exists = |- ?R. WeakOrder R *)

(* (3) Define the new type ('a)po as a conservative extension to HOL by: *)

val po_type_definition = new_type_definition ("po",WeakOrder_Exists);

(*     which produces:
      val po_type_definition =
      |- ?(rep :'a po -> 'a -> 'a -> bool).
           TYPE_DEFINITION (WeakOrder :('a -> 'a -> bool) -> bool) rep *)

(* (4) Prove that ('a)po is isomorphic to the set of partial orderings
       using the ml function define_new_type_bijections. The mapping
       functions to and from ('a)po to 'a are the abstraction (constructor)
       and representation (destructor) functions PO and repPO. *)

val po_bij = save_thm ("po_bij",
     (define_new_type_bijections
      {name="po_tybij", ABS="PO", REP="repPO", tyax=po_type_definition}));
```

```
(* po_bij =
    |- (!a. PO (repPO a) = a) /\ !r. WeakOrder r = (repPO (PO r) = r) *)

(* (5)  Prove additional theorems about the type ('a)po as follows: *)
(* Show that the the construction of ('a)po is one-to-one, i.e., unique. *)
val abs_po11 =
    save_thm
    ("abs_po11",
     (GEN_BETA_RULE (prove_abs_fn_one_one po_bij)));

(* abs_po11 =
 |- !r r'. WeakOrder r ==> WeakOrder r' ==> ((PO r = PO r') = (r = r')) *)

(* Show that every relation that is a partial order over 'a is in ('a)po *)
val onto_po =
    save_thm
    ("onto_po",
     (prove_rep_fn_onto po_bij));

(* onto_po = |- !r. WeakOrder r = ?a. r = repPO a *)

(* Show that every member of ('a)po is constructed from a
   partial order over 'a *)
val absPO_fn_onto =
    save_thm
    ("absPO_fn_onto",
     (prove_abs_fn_onto po_bij));

(*  absPO_fn_onto = |- !a. ?r. (a = PO r) /\ WeakOrder r *)

(* Get the explicit partial-order properties of the relation corresponding
   to anything of type ('a)po. *)
val [PO_repPO, WO_repPO] = map2 (fn x => fn y => save_thm (x, y))
                                ["PO_repPO", "WO_repPO"]
                                (CONJUNCTS po_bij);

(*  PO_repPO = |- !a. PO (repPO a)
    WO_repPO = |- !r. WeakOrder r = (repPO (PO r) = r) *)

val repPO_iPO_partial_order = save_thm ("repPO_iPO_partial_order",
    REWRITE_RULE
    [(SPEC (Term`iPO:'a po`) PO_repPO),
     WeakOrder, reflexive_def, transitive_def, antisymmetric_def]
    (SPEC (Term`(repPO iPO)`) WO_repPO));

(* repPO_iPO_partial_order =
    |- (!x. repPO iPO x x) /\
       (!x y. repPO iPO x y /\ repPO iPO y x ==> (x = y)) /\
       !x y z. repPO iPO x y /\ repPO iPO y z ==> repPO iPO x z *)


(*********************************************************************)
(* We now introduce (1) a trivial partial order, (2) the composition of*)
(* partial orders, and (3) show that subset is a partial order.  These *)
(* are used when creating a security and integrity levels using       *)
(* categories or compartments.                                         *)
(*********************************************************************)
(* First a trivial partial order, a product construction on partial
   orders, and definition of the superset partial order. *)

(*********************************************************************)
(* A partial order with one element: used for trivial orders if needed *)
(*********************************************************************)
val one_weakorder_def = Define `one_weakorder (x:one) (y:one) = T`;

val one_weakorder_WO =
```

```
   store_thm
   ("one_weakorder_WO",
    Term `WeakOrder one_weakorder`,
    REWRITE_TAC
    [one_weakorder_def, WeakOrder, reflexive_def,
     transitive_def,antisymmetric_def] THEN
    ONCE_REWRITE_TAC [one] THEN
    REPEAT GEN_TAC THEN
    REFL_TAC);

val O1_def =
    Define `O1 = PO one_weakorder`;

val repPO_O1 =
   store_thm
   ("repPO_O1",
    Term`repPO O1 = one_weakorder`,
    REWRITE_TAC
     [O1_def, po_bij,
      EQ_MP (ISPEC (Term`one_weakorder`) WO_repPO) one_weakorder_WO]);


(*********************************************************************)
(* We can create a partial order by composing two partial orders to  *)
(* to form a third.                                                  *)
(*********************************************************************)
(* RPROD, the product of two (Curried) binary relations, turns out to be
   already defined in pairTheory; it will be good style to use it. *)

(* RPROD_DEF = |- !R1 R2. RPROD R1 R2 = (\(s,t) (u,v). R1 s u /\ R2 t v) *)

val RPROD_THM =
   store_thm
   ("RPROD_THM",
    Term`!r s a b:'x#'y.
           RPROD r s a b = r (FST a) (FST b) /\ s (SND a) (SND b)`,
    REPEAT GEN_TAC THEN
    REWRITE_TAC [RPROD_DEF] THEN
    CONV_TAC (LAND_CONV (ONCE_DEPTH_CONV (REWR_CONV (GSYM PAIR)))) THEN
    CONV_TAC (DEPTH_CONV PAIRED_BETA_CONV) THEN REFL_TAC);

(* The following is perhaps a long-winded approach, but it seems
conceivably worth knowing that some individual properties of relations
are preserved by product: *)

val refl_prod_refl =
   store_thm
   ("refl_prod_refl",
    Term`!r:'a->'a->bool s:'b->'b->bool.
          reflexive r /\ reflexive s ==> reflexive (RPROD r s)`,
    REPEAT GEN_TAC THEN
    REWRITE_TAC [reflexive_def, RPROD_THM] THEN
    CONV_TAC (DEPTH_CONV PAIRED_BETA_CONV) THEN
    STRIP_TAC THEN ASM_REWRITE_TAC []);

val trans_prod_trans =
   store_thm
   ("trans_prod_trans",
    Term`!r:'a->'a->bool s:'b->'b->bool.
          transitive r /\ transitive s ==> transitive (RPROD r s)`,
    REPEAT GEN_TAC THEN
    REWRITE_TAC [transitive_def, RPROD_THM] THEN
    REPEAT STRIP_TAC THEN RES_TAC);

val antisym_prod_antisym =
   store_thm
   ("antisym_prod_antisym",
```

```
  Term `!r:'a->'a->bool s:'b->'b->bool.
        antisymmetric r /\ antisymmetric s ==> antisymmetric (RPROD r s)`,
  REPEAT GEN_TAC THEN
  REWRITE_TAC [antisymmetric_def, RPROD_THM] THEN
  ONCE_REWRITE_TAC [GSYM PAIR] THEN
  ONCE_REWRITE_TAC [PAIR_EQ] THEN
  REPEAT STRIP_TAC THEN
  RES_TAC);

val WO_prod_WO =
 store_thm
 ("WO_prod_WO",
  Term `!r:'a->'a->bool s:'b->'b->bool.
        WeakOrder r /\ WeakOrder s ==> WeakOrder (RPROD r s)`,
  REWRITE_TAC [WeakOrder] THEN
  REPEAT STRIP_TAC THENL
  [MATCH_MP_TAC refl_prod_refl,
   MATCH_MP_TAC antisym_prod_antisym,
   MATCH_MP_TAC trans_prod_trans] THEN
  ASM_REWRITE_TAC []);

val prod_PO_def =
 Define
 `prod_PO (PO1:'a po) (PO2:'b po) = PO (RPROD (repPO PO1) (repPO PO2))`;

val repPO_prod_PO =
 store_thm
 ("repPO_prod_PO",
  Term
  `!po1:'a po po2:'b po.
    repPO (prod_PO po1 po2) = RPROD (repPO po1) (repPO po2)`,
  REPEAT GEN_TAC THEN
  REWRITE_TAC [prod_PO_def, GSYM (CONJUNCT2 po_bij)] THEN
  MATCH_MP_TAC WO_prod_WO THEN
  REWRITE_TAC [WO_repPO, po_bij]);

(* Since inclusion is a weak order, we can define, for any type 'a,
   the partial order Subset_PO of type ('a -> bool) po. *)

val SUBSET_WO =
 store_thm
 ("SUBSET_WO",
  Term `WeakOrder ($SUBSET:('a -> bool) -> ('a -> bool) -> bool)`,
  REWRITE_TAC
  [WeakOrder, reflexive_def, antisymmetric_def, transitive_def,
   SUBSET_REFL, SUBSET_ANTISYM, SUBSET_TRANS]);

val Subset_PO_def =
 Define `Subset_PO:('a -> bool) po = PO $SUBSET`;

val repPO_Subset_PO =
 store_thm
 ("repPO_Subset_PO",
  Term
  `repPO Subset_PO = $SUBSET:('a->bool) -> ('a->bool) -> bool`,
  REWRITE_TAC
  [Subset_PO_def, po_bij,
   EQ_MP
   (ISPEC
    (Term `$SUBSET:('a->bool) -> ('a->bool) -> bool`)
    WO_repPO)
   SUBSET_WO]);


(*****************************************************
 * Datatypes "Princ", "IntLevel", "SecLevel", and "Form"
 *****************************************************)
```

```
(* Initial a's on some type variables below are a Hol kludge to make the
   polymorphic types take their type parameters in the order we expect.*)

val _ = Hol_datatype
    `Princ = Name of 'apn
           | meet of Princ => Princ
           | quoting of Princ => Princ;

     IntLevel = iLab of 'il
              | il of 'apn;

     SecLevel = sLab of 'sl
              | sl of 'apn`;

(* SKC: we add numerical expressions to Form 2/13/2009 *)
val _ = Hol_datatype
    `Form = TT
          | FF
          | prop of 'aavar
          | notf of Form
          | andf of Form => Form
          | orf of Form => Form
          | impf of Form => Form
          | eqf of Form => Form
          | says of 'apn Princ => Form
          | speaks_for of 'apn Princ => 'apn Princ
          | controls of 'apn Princ => Form
          | reps of 'apn Princ => 'apn Princ => Form
          | domi of ('apn, 'il) IntLevel => ('apn, 'il) IntLevel
          | eqi of ('apn, 'il) IntLevel => ('apn, 'il) IntLevel
          | doms of ('apn, 'sl) SecLevel => ('apn, 'sl) SecLevel
          | eqs of ('apn, 'sl) SecLevel => ('apn, 'sl) SecLevel
          | eqn of num => num
          | lte of num => num
          | lt of num => num`;

(* Change "meet" and "quoting" to infix operators *)

val _ = set_fixity "meet" (Infixr 630);
val _ = set_fixity "quoting" (Infixr 620);

(* and the rest *)

val _ = set_fixity "andf" (Infixr 580);
val _ = set_fixity "orf" (Infixr 570);
val _ = set_fixity "impf" (Infixr 560);
val _ = set_fixity "eqf" (Infixr 550);
val _ = set_fixity "says" (Infixr 590);
val _ = set_fixity "speaks_for" (Infixr 615);
val _ = set_fixity "controls" (Infixr 590);
val _ = set_fixity "domi" (Infixr 590);
val _ = set_fixity "eqi" (Infixr 590);
val _ = set_fixity "doms" (Infixr 590);
val _ = set_fixity "eqs" (Infixr 590);
val _ = set_fixity "eqn" (Infixr 590);
val _ = set_fixity "lte" (Infixr 590);
val _ = set_fixity "lt" (Infixr 590);

(* We want eventually to show soundness: that every derivable formula of
 a.c.l. holds in every world of every Kripke structure M. But derivability
 is now to be parameterized by partial orders Oi and Os; these must then
 be separate parameters to the semantic function also, so that the notion
 "all Kripke structures" does not entail a free choice of what the partial
 orders are.

 Thus the new Kripke structure, besides its I and J components, has just 2
```

```
 mappings assigning integrity and security levels to the principal names.*)

val _ = Hol_datatype
    `Kripke = KS of ('aavar -> ('aaworld set)) =>
                    ('apn -> ('aaworld -> ('aaworld set))) =>
                    ('apn -> 'il) => ('apn -> 'sl)`;

(* type arguments to Kripke, in left-to-right order:
   propvar_type, world_type, princ_name_type, integ_type, sec_type *)

(**********
* accessor functions for Kripke structures
**********)
val intpKS_def =
    Define `intpKS(KS Intp Jfn ilmap slmap) = Intp`;

val jKS_def =
    Define `jKS(KS Intp Jfn ilmap slmap) = Jfn`;

val imapKS_def =
    Define `imapKS(KS Intp Jfn ilmap slmap) = ilmap`;

val smapKS_def =
    Define `smapKS(KS Intp Jfn ilmap slmap) = slmap`;

(*********
* properties of Kripke destructors
*********)

val KS_bij =
store_thm(
        "KS_bij",
        ``!M.M = KS (intpKS M)(jKS M)(imapKS M)(smapKS M)``,
        Induct_on `M` THEN
        REWRITE_TAC [intpKS_def, jKS_def, imapKS_def, smapKS_def]);

val _ = export_theory();

end;
```

D.2 aclsemantics Theory

```
(* added semantics of arithmetic expressions, Aexp, 2/12/2009 *)
(* mutilated by lm, 1/24/09, and eqp removed 1/29/09 *)
(*=======================================================*)
(* We build the semantic definitions of the access-control logic *)
(*=======================================================*)
(*
load "aclfoundationTheory";
*)
structure aclsemanticsScript =
struct

open HolKernel boolLib Parse bossLib;
open aclfoundationTheory relationTheory;

val _ = new_theory "aclsemantics";

(************************************************************
* Define the extended mapping Jext that maps principal expressions
* Princ to a relation on 'ws. It is parameterized on
* J:PName -> ('w -> 'w set).
************************************************************)
val Jext_def =
    Define
```

```
    `(Jext (J:'pn -> 'w ->'w set) (Name s) = J s) /\
     (Jext J (P1 meet P2) = ((Jext J P1) RUNION (Jext J P2))) /\
     (Jext J (P1 quoting P2) = (Jext J P2) O (Jext J P1))`;

val Jext_names = ["name_def", "meet_def", "quoting_def"];
val _ = map2 (fn x=>fn y=>save_thm(x,y)) Jext_names (CONJUNCTS Jext_def);

(*************************************************************
* Define the mapping from IntLevel to integrity levels at the
* semantic level. This function is Lifn.
*************************************************************)
val Lifn_def =
    Define
    `(Lifn M (iLab l) = l) /\
     (Lifn M (il name) = imapKS M name)`;

(*************************************************************
* Define the mapping from SecLevel to security levels at the
* semantic level. This function is Lsfn.
*************************************************************)
val Lsfn_def =
    Define
    `(Lsfn M (sLab l) = l) /\
     (Lsfn M (sl name) = smapKS M name)`;

(*************************************************************
* Define the semantic meaning function Efn. Efn is parameterized
* on partial orders Oi for integrity levels and Os for security levels
* and on Kripke structure
*                            KS Intrp Jfn imap smap
*
* UNIV:('w)set corresponds to W in the Kripke structure described
* in our book. UNIV is non-empty, as every type in HOL has at least
* one member.
*************************************************************)
val Efn_def =
    Define
    `(Efn (Oi:'il po) (Os:'is po) (M:('w,'v,'pn,'il,'is) Kripke)
                 TT = UNIV) /\
     (Efn Oi Os M FF = {}) /\
     (Efn Oi Os M (prop p) = ((intpKS M) p)) /\
     (Efn Oi Os M (notf f) = (UNIV DIFF (Efn Oi Os M f))) /\
     (Efn Oi Os M (f1 andf f2) =
           ((Efn Oi Os M f1) INTER (Efn Oi Os M f2))) /\
     (Efn Oi Os M (f1 orf f2) =
           ((Efn Oi Os M f1) UNION (Efn Oi Os M f2))) /\
     (Efn Oi Os M (f1 impf f2) =
           ((UNIV DIFF (Efn Oi Os M f1)) UNION (Efn Oi Os M f2)))  /\
     (Efn Oi Os M (f1 eqf f2) =
           ((UNIV DIFF (Efn Oi Os M f1) UNION (Efn Oi Os M f2)) INTER
            (UNIV DIFF (Efn Oi Os M f2) UNION (Efn Oi Os M  f1)))) /\
     (Efn Oi Os M(P says f) =
           {w | Jext (jKS M) P w SUBSET (Efn Oi Os M f)}) /\
     (Efn Oi Os M (P speaks_for Q) =
           (if ((Jext (jKS M) Q) RSUBSET (Jext (jKS M) P)) then UNIV else
           {})) /\
     (Efn Oi Os M(P controls f) =
           ((UNIV DIFF
           ({w | Jext (jKS M) P w SUBSET Efn Oi Os M f})) UNION
            (Efn Oi Os M f)))  /\
     (Efn Oi Os M (reps P Q f) =
           ((UNIV DIFF
           ({w | Jext (jKS M) (P quoting Q) w SUBSET
                 Efn Oi Os M f})) UNION
            {w | Jext (jKS M) Q w SUBSET Efn Oi Os M f})) /\
     (Efn Oi Os M (intl1 domi intl2) = (* note inversion 3/12/09 *)
           (if repPO Oi (Lifn M intl2) (Lifn M intl1)
```

```
            then UNIV else {})) /\
      (Efn Oi Os M (intl2 eqi intl1) = (* ** note inversion 7/30/09 ** *)
            (if repPO Oi (Lifn M intl2) (Lifn M intl1)
            then UNIV else {}) INTER
            (if repPO Oi (Lifn M intl1) (Lifn M intl2)
            then UNIV else {})) /\
      (Efn Oi Os M (secl1 doms secl2) = (* note inversion *)
            (if repPO Os (Lsfn M secl2) (Lsfn M secl1)
            then UNIV else {})) /\
      (Efn Oi Os M (secl2 eqs secl1) = (* ** note inversion ** *)
            (if repPO Os (Lsfn M secl2) (Lsfn M secl1)
            then UNIV else {}) INTER
            (if repPO Os (Lsfn M secl1) (Lsfn M secl2)
            then UNIV else {})) /\
      (Efn Oi Os M ((numExp1:num) eqn (numExp2:num)) =
            (if (numExp1 = numExp2)
             then UNIV else {})) /\
      (Efn Oi Os M ((numExp1:num) lte (numExp2:num)) =
            (if (numExp1 <= numExp2)
             then UNIV else {})) /\
      (Efn Oi Os M ((numExp1:num) lt (numExp2:num)) =
            (if (numExp1 < numExp2)
             then UNIV else {}))`;


(*********
* save each definition of Efn individually
*********)

val Efn_names = ["TT_def", "FF_def", "prop_def", "notf_def", "andf_def",
                 "orf_def", "impf_def", "eqf_def", "says_def",
                 "speaks_for_def", "controls_def", "reps_def",
                 "domi_def", "eqi_def", "doms_def", "eqs_def", "eqn_def", "lte_def", "lt_def"];

val _ = map2 (fn x => fn y => save_thm(x,y)) Efn_names (CONJUNCTS Efn_def);

(************************
* Syntactic Sugar Properties
* This is to make sure that what we've defined here
* matches what we have in the textbook.
************************)

(*********
* Fetch the individual theorems needed
*********)
val eqf_def = DB.fetch "aclsemantics" "eqf_def";
val andf_def = DB.fetch "aclsemantics" "andf_def";
val impf_def = DB.fetch "aclsemantics" "impf_def";
val controls_def = DB.fetch "aclsemantics" "controls_def";
val says_def = DB.fetch "aclsemantics" "says_def";
val eqi_def = DB.fetch "aclsemantics" "eqi_def";
val domi_def = DB.fetch "aclsemantics" "domi_def";
val eqs_def = DB.fetch "aclsemantics" "eqs_def";
val doms_def = DB.fetch "aclsemantics" "doms_def";

val eqf_impf =
store_thm(
        "eqf_impf",
        ``!M f1 f2.
        Efn Oi Os M (f1 eqf f2) =
        Efn Oi Os M((f1 impf f2) andf (f2 impf f1))``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [eqf_def, andf_def, impf_def]);

val controls_says =
store_thm(
        "controls_says",
```

```
        ``!M P f.
        Efn Oi Os M (P controls f) =
        Efn Oi Os M ((P says f) impf f)``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [controls_def, impf_def, says_def]);

val eqi_domi =
store_thm(
        "eqi_domi",
        ``!M intL1 intL2.
        Efn Oi Os M (intL1 eqi intL2) =
        Efn Oi Os M ((intL2 domi intL1) andf (intL1 domi intL2))``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [eqi_def,andf_def,domi_def]);

val eqs_doms =
store_thm(
        "eqs_doms",
        ``!M secL1 secL2.
        Efn Oi Os M (secL1 eqs secL2) =
        Efn Oi Os M ((secL2 doms secL1) andf (secL1 doms secL2))``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [eqs_def,andf_def,doms_def]);

(*******************
* Export the theory
*******************)
val _ = print_theory "-";
val _ = export_theory();

end;
```

D.3 aclrules Theory

```
(* modified by SKC, 11/9/2011
   - added AndSays_Eq
   modified by SKC, 2/19/2009
   - proved all core rules and many derived ones. *)
(* mutilated by lm, 1/24/09  *)
(*=========================================================*)
(* We build the semantic definitions of the access-control logic *)
(*=========================================================*)
(*
load "aclsemanticsTheory";
*)
structure aclrulesScript =
struct

open HolKernel boolLib Parse bossLib;
open pred_setLib pred_setTheory;
open aclfoundationTheory aclsemanticsTheory relationTheory;

val _ = new_theory "aclrules";

(*********
* The definition of M |= f, pronounced "M satisfies f", where M is
* a Kripke structure and f is a formula in the access-control logic.
* We say, M satisfies f, whenever f is true in all worlds of M. This
* relation is denoted by ``M sat f``, as defined below.
*********)
val _ = set_fixity "sat" (Infixr 540);
val sat_def =
    Define
    `(M, Oi, Os) sat f = ((Efn Oi Os M f) = UNIV:('world) set)`;
```

```
(*********
* A property of Images
*********)
val world_says =
    store_thm
    ("world_says",
     Term`!M Oi Os P f w.
     w IN Efn Oi Os M (P says f) =
        !v. v IN Jext (jKS M) P w ==> v IN Efn Oi Os M f`,
     REPEAT GEN_TAC THEN REWRITE_TAC [says_def] THEN
     CONV_TAC (LAND_CONV SET_SPEC_CONV) THEN
     REWRITE_TAC [SUBSET_DEF]);

val cond_lemma =
TAC_PROOF(
        ([],``! b t1 t2.(~ (t2:'a= t1:'a))==>
                        ((if b then t1 else t2) = t1) ==> b``),
        REPEAT GEN_TAC THEN
        BOOL_CASES_TAC ``b:bool`` THEN
        REWRITE_TAC []);

val [repPO_iPO_reflexive, repPO_iPO_antisymmetric, repPO_iPO_transitive] =
    CONJUNCTS repPO_iPO_partial_order;

(******
* Properties of sat
******)

val sat_allworld =
store_thm(
        "sat_allworld",
        ``!M f. (M, Oi, Os) sat f = !w. w IN Efn Oi Os M f``,
        REWRITE_TAC [sat_def, UNIV_DEF, IN_DEF] THEN
        CONV_TAC
        (DEPTH_CONV FUN_EQ_CONV THENC DEPTH_CONV BETA_CONV) THEN
        REWRITE_TAC [TT_def, IN_UNIV]);

(******
* Properties of propositions:
******)

val world_T =
    store_thm
    ("world_T",
    ``!M Oi Os w. w IN Efn Oi Os M TT``,
    REWRITE_TAC [TT_def, IN_UNIV]);

val world_F =
    store_thm
    ("world_F",
    ``!M Oi Os w.~ (w IN Efn Oi Os M FF)``,
    REWRITE_TAC [FF_def, NOT_IN_EMPTY]);

val world_not =
    store_thm
    ("world_not",
    ``!M Oi Os f w.w IN Efn Oi Os M (notf f) = ~ (w IN Efn Oi Os M f)``,
    REWRITE_TAC [notf_def, IN_DIFF, IN_UNIV]);

val world_not =
    store_thm
    ("world_not",
    ``!M Oi Os f w.w IN Efn Oi Os M (notf f) = ~ (w IN Efn Oi Os M f)``,
    REWRITE_TAC [notf_def, IN_DIFF, IN_UNIV]);

val world_not =
    store_thm
```

```
    ("world_not",
    ``!M Oi Os f w.w IN Efn Oi Os M (notf f) = ~ (w IN Efn Oi Os M f)``,
    REWRITE_TAC [notf_def, IN_DIFF, IN_UNIV]);

val world_and =
    store_thm
    ("world_and",
    ``!M Oi Os f1 f2 w.
    w IN Efn Oi Os M (f1 andf f2) = w IN Efn Oi Os M f1 /\
    w IN Efn Oi Os M f2``,
    REWRITE_TAC [andf_def, IN_INTER]);

val world_or =
    store_thm
    ("world_or",
    ``!M f1 f2 w.
    w IN Efn Oi Os M (f1 orf f2) = w IN Efn Oi Os M f1 \/
    w IN Efn Oi Os M f2``,
    REWRITE_TAC [orf_def, IN_UNION]);

val world_imp =
    store_thm
    ("world_imp",
    ``!M Oi Os f1 f2 w.
    w IN Efn Oi Os M (f1 impf f2) = w IN Efn Oi Os M f1 ==>
    w IN Efn Oi Os M f2``,
    REWRITE_TAC [impf_def, IN_DIFF, IN_UNION, IN_UNIV, IMP_DISJ_THM]);

val world_eq =
    store_thm
    ("world_eq",
    ``!M Oi Os f1 f2 w.
    w IN Efn Oi Os M (f1 eqf f2) = (w IN Efn Oi Os M f1 =
    w IN Efn Oi Os M f2)``,
    REPEAT GEN_TAC
    THEN REWRITE_TAC [eqf_def, IN_DIFF, IN_UNION, IN_INTER, IN_UNIV]
    THEN CONV_TAC (RAND_CONV (REWRITE_CONV [EQ_IMP_THM, IMP_DISJ_THM]))
    THEN REFL_TAC);

val world_eqn =
    store_thm
    ("world_eqn",
     ``!M Oi Os n1 n2 w.
      w IN Efn Oi Os m (n1 eqn n2) = (n1 = n2)``,
   REPEAT GEN_TAC THEN
   REWRITE_TAC [eqn_def] THEN
   COND_CASES_TAC THEN
   REWRITE_TAC [IN_UNIV, NOT_IN_EMPTY]);

val world_lte =
    store_thm
    ("world_lte",
     ``!M Oi Os n1 n2 w.
      w IN Efn Oi Os m (n1 lte n2) = (n1 <= n2)``,
   REPEAT GEN_TAC THEN
   REWRITE_TAC [lte_def] THEN
   COND_CASES_TAC THEN
   REWRITE_TAC [IN_UNIV, NOT_IN_EMPTY]);

val world_lt =
    store_thm
    ("world_lt",
     ``!M Oi Os n1 n2 w.
      w IN Efn Oi Os m (n1 lt n2) = (n1 < n2)``,
   REPEAT GEN_TAC THEN
   REWRITE_TAC [lt_def] THEN
   COND_CASES_TAC THEN
```

```
   REWRITE_TAC [IN_UNIV, NOT_IN_EMPTY]);

(*********************************
* INFERENCE RULES
* Our inference rules in the textbook are written as
*
*      H1 ... Hn
*      -------- Rule Name
*         C
*
* Inference rules are theorems:
* !(M:('a IntLabel, 'b SecLabel, 'world) Kripke)
*    (H1:('a IntLabel, 'b SecLabel) Form) .. Hn.
*   M sat H1 ==> M sat H2 ==> ... ==> M sat Hn ==> M sat C
*********************************)



(*****************************************************************
* A tactic to prove goals of the form M sat <instance of tautology> *
*****************************************************************)
val ACL_TAUT_TAC =
    REWRITE_TAC
    [sat_allworld, world_T, world_F, world_not,
     world_and, world_or, world_imp, world_eq]
    THEN DECIDE_TAC;

(******************
* reflexivity of domi
*
*    --------------- Reflexivity of domi
*      intL domi intL
******************)
val domi_reflexive =
store_thm(
        "domi_reflexive",
        ``!M Oi Os l. (M, Oi, Os) sat (l domi l)``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [sat_def, domi_def, repPO_iPO_partial_order]);

(******************
* transitivity of domi
*
* intL1 domi intL2     intL2 domi intL3
* ------------------------------------ Transitivity of domi
*                intL1 domi intL3
******************)
val domi_transitive =
store_thm(
        "domi_transitive",
        ``!M Oi Os l1 l2 l3.
            ((M,Oi,Os) sat (l1 domi l2)) ==>
            ((M,Oi,Os) sat (l2 domi l3)) ==>
            ((M,Oi,Os) sat (l1 domi l3))``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [sat_def, domi_def, repPO_iPO_partial_order] THEN
        REPEAT DISCH_TAC THEN
        IMP_RES_TAC (MATCH_MP cond_lemma EMPTY_NOT_UNIV) THEN
        IMP_RES_TAC repPO_iPO_transitive THEN
        ASM_REWRITE_TAC []);


(*********
* antisymmetry of domi
*
* intL1 domi intL2     intL2 domi intL1
* ------------------------------------ Antisymmetry of domi
```

```
*                  intL1 eqi intL2
**********)
val domi_antisymmetric =
store_thm(
        "domi_antisymmetric",
        ``!M Oi Os l1 l2.
            ((M,Oi,Os) sat (l1 domi l2)) ==>
            ((M,Oi,Os) sat (l2 domi l1)) ==>
            ((M,Oi,Os) sat (l1 eqi l2))``,
            REPEAT GEN_TAC THEN
            REWRITE_TAC [sat_def, domi_def, eqi_def,
                         repPO_iPO_partial_order] THEN
            REPEAT DISCH_TAC THEN
            ASM_REWRITE_TAC [INTER_UNIV]);

(****************
* eqi_Eq
* (l1 eqi l2) = (l2 domi l1 andf l1 domi l2)
****************)
val eqi_Eq =
store_thm
   ("eqi_Eq",
    ``!M Oi Os l1 l2.((M,Oi,Os) sat l1 eqi l2) =
                     ((M,Oi,Os) sat (l2 domi l1) andf (l1 domi l2))``,
    REPEAT GEN_TAC THEN
    REWRITE_TAC [sat_def,eqi_domi]);

(******************
* reflexivity of doms
*
*    ---------------- Reflexivity of doms
*      secL doms secL
******************)
val doms_reflexive =
store_thm(
        "doms_reflexive",
        ``!M Oi Os l. (M,Oi,Os) sat (l doms l)``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [sat_def, doms_def, repPO_iPO_reflexive]);


(******************
* transitivity of doms
*
* secL1 doms secL2     secL2 doms secL3
* ------------------------------------- Transitivity of doms
*                 secL1 doms secL3
******************)
val doms_transitive =
store_thm(
        "doms_transitive",
        ``!M Oi Os l1 l2 l3.
            ((M,Oi,Os) sat (l1 doms l2)) ==>
            ((M,Oi,Os) sat (l2 doms l3)) ==>
            ((M,Oi,Os) sat (l1 doms l3))``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [sat_def, doms_def, repPO_iPO_partial_order] THEN
        REPEAT DISCH_TAC THEN
        IMP_RES_TAC (MATCH_MP cond_lemma EMPTY_NOT_UNIV) THEN
        IMP_RES_TAC repPO_iPO_transitive THEN
        ASM_REWRITE_TAC []);

(**********
* antisymmetry of doms
*
* secL1 domi secL2     secL2 domi secL1
* ------------------------------------- Antisymmetry of doms
```

```
*                    secL1 eqs secL2
*********)
val doms_antisymmetric =
store_thm(
        "doms_antisymmetric",
        ``!M Oi Os l1 l2.
            ((M,Oi,Os) sat (l1 doms l2)) ==>
            ((M,Oi,Os) sat (l2 doms l1)) ==>
            ((M,Oi,Os) sat (l1 eqs l2))``,
            REPEAT GEN_TAC THEN
            REWRITE_TAC [sat_def, doms_def, eqs_def,
                         repPO_iPO_partial_order] THEN
            REPEAT DISCH_TAC THEN
            ASM_REWRITE_TAC [INTER_UNIV]);

(****************
* eqs_Eq
*
* (l1 eqs l2) = (l2 doms l1 andf l1 doms l2)
****************)
val eqs_Eq =
store_thm
   ("eqs_Eq",
    ``!M Oi Os l1 l2.((M,Oi,Os) sat l1 eqs l2) =
                     ((M,Oi,Os) sat (l2 doms l1) andf (l1 doms l2))``,
    REPEAT GEN_TAC THEN
    REWRITE_TAC [sat_def,eqs_doms]);

(**************************
* Modus Ponens
*
* f1   f1 impf f2
* -------------- Modus Ponens
*       f2
**************************)
val Modus_Ponens =
store_thm(
        "Modus_Ponens",
        ``!M Oi Os f1 f2.
            ((M,Oi,Os) sat f1) ==>
            ((M,Oi,Os) sat (f1 impf f2)) ==>
            ((M,Oi,Os) sat f2)``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [sat_def, impf_def] THEN
        DISCH_TAC THEN
        ASM_REWRITE_TAC [DIFF_UNIV, UNION_EMPTY]);

(**************************
* Says
*
*           f
* -------------- Says
*     P says f
**************************)
val Says =
store_thm(
        "Says",
        ``!M Oi Os P f.
            ((M,Oi,Os) sat f) ==>
            ((M,Oi,Os) sat (P says f))``,
        REPEAT GEN_TAC THEN
        REWRITE_TAC [sat_def, says_def] THEN
        DISCH_TAC THEN
        ASM_REWRITE_TAC [SUBSET_UNIV, GSPEC_T]);

(************************
* MP Says
```

```
*
* ──────────────────────────────────────────────
* P says (f1 impf f2) impf ((P says f1) impf (P says f2))
***************************)
val MP_Says =
    store_thm
    ("MP_Says",
    Term`!M Oi Os P f1 f2.
    (M,Oi,Os) sat ((P says (f1 impf f2)) impf
                  ((P says f1) impf (P says f2)))`,
    REPEAT GEN_TAC THEN
    REWRITE_TAC [sat_allworld, world_says, world_imp] THEN
    REPEAT STRIP_TAC THEN
    RES_TAC);

(***************************
* Speaks For
*
* ──────────────────────────────────────────────
* (P speaks_for Q) impf ((P says f) impf (Q says f))
***************************)

(**properties of sets and images **)
val UNIV_DIFF_SUBSET =
store_thm
  ("UNIV_DIFF_SUBSET",
  ``!R1 R2. (R1 SUBSET R2) ==> (((UNIV DIFF R1) UNION R2) = UNIV)``,
  REWRITE_TAC
  [SUBSET_DEF, DIFF_DEF, IN_UNIV, UNION_DEF] THEN
  CONV_TAC (DEPTH_CONV SET_SPEC_CONV) THEN
  REPEAT GEN_TAC THEN
  REWRITE_TAC [SPECL [``x IN R1``, ``x IN R2``] IMP_DISJ_THM] THEN
  DISCH_TAC THEN
  ASM_REWRITE_TAC [GSPEC_T]);

val Image_SUBSET =
store_thm
  ("Image_SUBSET",
   ``!R1 R2.(R2 RSUBSET R1) ==> !w.((R2 w) SUBSET (R1 w))``,
   REWRITE_TAC [RSUBSET, SUBSET_DEF, IN_DEF] THEN
   CONV_TAC(DEPTH_CONV BETA_CONV) THEN
   PROVE_TAC []);

val SUBSET_Image_SUBSET =
store_thm
  ("SUBSET_Image_SUBSET",
   ``!R1 R2 R3.
   (!w1.((R2 w1) SUBSET (R1 w1))) ==>
   (!w.({w | (R1 w) SUBSET R3} SUBSET {w | (R2 w) SUBSET R3}))``,
   REWRITE_TAC [SUBSET_DEF] THEN
   CONV_TAC(DEPTH_CONV SET_SPEC_CONV) THEN
   PROVE_TAC []);

val speaks_for_SUBSET =
store_thm
  ("speaks_for_SUBSET",
  ``!R3 R2 R1.(R2 RSUBSET R1) ==>
  (!w.({w | (R1 w) SUBSET R3} SUBSET {w | (R2 w) SUBSET R3}))``,
  REPEAT STRIP_TAC THEN
  IMP_RES_TAC Image_SUBSET THEN
  IMP_RES_TAC SUBSET_Image_SUBSET THEN
  ASM_REWRITE_TAC []);

val UNIV_DIFF_SUBSET_lemma =
TAC_PROOF(
   ([],
     ``!M Oi Os P Q f. (Jext (jKS (M :('a, 'b, 'c, 'd, 'e) Kripke))
```

```
                          (Q :'c Princ) RSUBSET
          Jext (jKS M) (P :'c Princ)) ==> ((UNIV :'b -> bool) DIFF
        {(w :'b) |
          Jext (jKS (M :('a, 'b, 'c, 'd, 'e) Kripke)) (P :'c Princ) w SUBSET
          Efn (Oi :'d po) (Os :'e po) M (f :('a, 'c, 'd, 'e) Form)} UNION
        {w | Jext (jKS M) (Q :'c Princ) w SUBSET Efn Oi Os M f} =
        (UNIV :'b -> bool))``),
  REPEAT GEN_TAC THEN
  DISCH_THEN
  (fn th =>
    ASSUME_TAC
     (SPEC_ALL
     (MP
      (ISPEC
     ``Jext (jKS (M :('a, 'b, 'c, 'd, 'e) Kripke)) P``
     (ISPEC ``(Jext (jKS (M :('a, 'b, 'c, 'd, 'e) Kripke)) Q)``
     (ISPEC ``Efn (Oi :'d po) (Os :'e po) M (f :('a, 'c, 'd, 'e) Form)``
            speaks_for_SUBSET)))
     th))) THEN
    IMP_RES_TAC
     (ISPEC
     ``{(w:'b) | Jext (jKS M) (Q :'c Princ) w SUBSET Efn Oi Os M f}``
       (ISPEC ``{(w :'b) |
         Jext (jKS (M :('a, 'b, 'c, 'd, 'e) Kripke)) (P :'c Princ) w SUBSET
         Efn (Oi :'d po) (Os :'e po) M (f :('a, 'c, 'd, 'e) Form)}``
      UNIV_DIFF_SUBSET)));

val Speaks_For =
store_thm
   ("Speaks_For",
   ``!M Oi Os P Q f.
   (M,Oi,Os) sat (P speaks_for Q) impf ((P says f) impf (Q says f))``,
   REPEAT GEN_TAC THEN
   REWRITE_TAC [sat_def,says_def, impf_def, speaks_for_def] THEN
   COND_CASES_TAC THEN
   REWRITE_TAC [DIFF_EMPTY,UNION_UNIV, DIFF_UNIV, UNION_EMPTY] THEN
   IMP_RES_TAC UNIV_DIFF_SUBSET_lemma THEN
   ASM_REWRITE_TAC []);

(*************************
* Trans_Speaks_For
*
* P speaks_for Q     Q speaks_for R
* ------------------------------
*            P speaks_for R
*************************)
val RSUBSET_TRANS =
(hd o tl o tl)(CONJUNCTS(REWRITE_RULE
               [WeakOrder, transitive_def] RSUBSET_WeakOrder));

val Trans_Speaks_For =
store_thm
  ("Trans_Speaks_For",
  ``!M Oi Os P Q R.
      ((M,Oi,Os) sat P speaks_for Q) ==>
      ((M,Oi,Os) sat Q speaks_for R) ==>
      ((M,Oi,Os) sat P speaks_for R)``,
  REPEAT GEN_TAC THEN
  REWRITE_TAC [sat_def, speaks_for_def] THEN
  REPEAT
  (COND_CASES_TAC THEN
   REWRITE_TAC [EMPTY_NOT_UNIV]) THEN
  IMP_RES_TAC RSUBSET_TRANS);

(*************************
* Idemp_Speaks_For
*
```

```
* ----------------
* P speaks_for P
**************************)

val RSUBSET_REFL =
hd(CONJUNCTS (REWRITE_RULE [WeakOrder, transitive_def,
                            reflexive_def] RSUBSET_WeakOrder));

val Idemp_Speaks_For =
store_thm
  ("Idemp_Speaks_For",
   ``!M Oi Os P. (M,Oi,Os) sat (P speaks_for P)``,
   REPEAT GEN_TAC THEN
   REWRITE_TAC [sat_def, speaks_for_def] THEN
   COND_CASES_TAC THEN
   REWRITE_TAC [EMPTY_NOT_UNIV] THEN
   UNDISCH_TAC
   ``~(Jext (jKS (M :('a, 'b, 'c, 'd, 'e) Kripke)) (P :'c Princ) RSUBSET
     Jext (jKS M) P)`` THEN
   REWRITE_TAC [RSUBSET_REFL]);

(*************************
* Mono_speaks_for
*
*  P speaks_for P'        Q speaks_for Q'
* ----------------------------------------
*(P quoting Q) speaks_for (P' quoting Q')
*************************)

val Mono_speaks_for =
store_thm
  ("Mono_speaks_for",
    ``!M Oi Os P P' Q Q'.
      ((M,Oi,Os) sat (P speaks_for P')) ==>
       ((M,Oi,Os) sat (Q speaks_for Q')) ==>
      ((M,Oi,Os) sat ((P quoting Q) speaks_for (P' quoting Q')))``,
  REPEAT GEN_TAC THEN
  REWRITE_TAC
  [sat_def, quoting_def, speaks_for_def, RSUBSET, O_DEF] THEN
  PROVE_TAC []);


(************************
* And_Says
*
* -------------------------------------------------------
* ((P meet Q) says f) impf ((P says f) andf (Q says f))
*************************)
val Image_UNION =
store_thm
  ("Image_UNION",
   ``!R1 R2 w. (R1 RUNION R2) w = (R1 w) UNION (R2 w) ``,
   REPEAT GEN_TAC THEN
   REWRITE_TAC [RUNION, UNION_DEF, IN_DEF] THEN
   CONV_TAC (DEPTH_CONV (BETA_CONV)) THEN
   REWRITE_TAC [SYM (SPEC_ALL RUNION)] THEN
   REWRITE_TAC[SYM(ISPECL [``(R1 RUNION R2) w``,``x:'b``]
                SPECIFICATION), GSPEC_ID]);

val and_says_lemma =
store_thm
  ("and_says_lemma",
  ``!M Oi Os P Q f. (M,Oi,Os) sat ((P meet Q) says f) impf
                    ((P says f) andf (Q says f))``,
 REPEAT GEN_TAC THEN
 REWRITE_TAC
 [sat_def, meet_def, says_def, impf_def, andf_def,
```

```
  Image_UNION, UNION_SUBSET, INTER_DEF] THEN
 CONV_TAC(DEPTH_CONV SET_SPEC_CONV) THEN
 REWRITE_TAC [(SYM (SPEC_ALL COMPL_DEF)),COMPL_CLAUSES]);

val says_and_lemma =
store_thm
  ("says_and_lemma",
  ``!M Oi Os P Q f. (M,Oi,Os) sat ((P says f) andf (Q says f)) impf
                            ((P meet Q) says f)``,
 REPEAT GEN_TAC THEN
 REWRITE_TAC
 [sat_def, meet_def, says_def, impf_def, andf_def,
  Image_UNION, UNION_SUBSET, INTER_DEF] THEN
 CONV_TAC(DEPTH_CONV SET_SPEC_CONV) THEN
 REWRITE_TAC [(SYM (SPEC_ALL COMPL_DEF)),COMPL_CLAUSES]);

val And_Says =
store_thm
  ("And_Says",
   ``!M Oi Os P Q f. (M,Oi,Os) sat ((P meet Q) says f) eqf
                     ((P says f) andf (Q says f))``,
   REPEAT GEN_TAC THEN
   REWRITE_TAC
   [sat_def,eqf_impf,andf_def,
    (REWRITE_RULE[sat_def] and_says_lemma),
    (REWRITE_RULE[sat_def] says_and_lemma),
    INTER_UNIV]);

(****************************
* Quoting
*
* ---------------------------------------------------
* ((P meet Q) says f) eqf ((P says f) andf (Q says f))
*
****************************)
(********************
* Relating eqf and impf
********************)
val eqf_and_impf =
store_thm
  ("eqf_and_impf",
   ``!M Oi Os f1 f2.
     ((M,Oi,Os) sat (f1 eqf f2)) =
     ((M,Oi,Os) sat ((f1 impf f2) andf (f2 impf f1)))``,
   REWRITE_TAC [sat_def, eqf_impf]);

(******************
* Relating eqf and (M,Oi,Os) sat
******************)
val eqf_sat =
store_thm(
   "eqf_sat",
   ``!M Oi Os f1 f2.(M,Oi,Os) sat f1 eqf f2 ==>
                    ((M,Oi,Os) sat f1 = (M,Oi,Os) sat f2)``,
   PROVE_TAC [sat_allworld,world_eq]);

(* ------------------------------------------------------------------------- *)
(* Theorems dealing with equivalence and substitution of terms in           *)
(* the access-control logic                                                  *)
(*                                                                           *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* When the intersection of two sets is the universe, then each set is      *)
(* also the universe                                                         *)
(* ------------------------------------------------------------------------- *)
```

```
val INTER_EQ_UNIV =
TAC_PROOF(([],``((s:'a set) INTER t = univ(:'a)) = ((s = univ(:'a)) /\ (t = univ(:'a)))``),
PROVE_TAC[(GSYM EQ_UNIV),IN_INTER])

val _ = save_thm("INTER_EQ_UNIV",INTER_EQ_UNIV)

(* ------------------------------------------------------------------------ *)
(* conjunction in ACL related to conjunction in HOL                          *)
(* (M,Oi,Os) sat (f1 andf f2) = (M,Oi,Os) sat f1 /\ (M,Oi,Os) sat f2         *)
(*                                                                            *)
(* ------------------------------------------------------------------------ *)

val sat_andf_eq_and_sat =
TAC_PROOF(([],``((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f1 andf f2) =
  (((M,Oi,Os) sat f1) /\ ((M,Oi,Os) sat f2))``),
REWRITE_TAC[sat_def,world_and,andf_def,INTER_EQ_UNIV])

val _ = save_thm("sat_andf_eq_and_sat",sat_andf_eq_and_sat)

val DIFF_UNIV_SUBSET =
TAC_PROOF(([],``((univ(:'a) DIFF s) UNION t = univ(:'a)) = s SUBSET t``),
REWRITE_TAC[SET_EQ_SUBSET] THEN
REWRITE_TAC[SUBSET_DEF, DIFF_DEF, IN_UNIV, UNION_DEF] THEN
CONV_TAC (DEPTH_CONV SET_SPEC_CONV) THEN
REWRITE_TAC[SPECL [``x IN s``, ``x IN t``] IMP_DISJ_THM])

val _ = save_thm("DIFF_UNIV_SUBSET",DIFF_UNIV_SUBSET)



(* ------------------------------------------------------------------------ *)
(* The key theorem: Efn Oi Os (f1 eqf f2) = univ(:'b) =                      *)
(*                  (Efn Oi Os f1 = Efn Oi Os f2)                            *)
(*                                                                            *)
(* ------------------------------------------------------------------------ *)

val eqf_eq_lemma1 =
TAC_PROOF(([],
``((Efn (Oi:'d po) (Os:'e po) (M:('a,'b,'c,'d,'e)Kripke) f1) =
   (Efn (Oi:'d po) (Os:'e po) (M:('a,'b,'c,'d,'e)Kripke) f2)) ==>
  ((Efn Oi Os M (f1 eqf f2)) = univ(:'b))``),
REWRITE_TAC[eqf_def] THEN
DISCH_THEN (fn th => REWRITE_TAC[th,INTER_IDEMPOT,(MATCH_MP UNION_DIFF (SPEC_ALL SUBSET_UNIV))]))

val eqf_eq_lemma2 =
TAC_PROOF(
([],
``((Efn Oi Os M (f1 eqf f2)) = univ(:'b)) ==>
  ((Efn (Oi:'d po) (Os:'e po) (M:('a,'b,'c,'d,'e)Kripke) f1) =
   (Efn (Oi:'d po) (Os:'e po) (M:('a,'b,'c,'d,'e)Kripke) f2))``),
REWRITE_TAC[eqf_def,INTER_EQ_UNIV,DIFF_UNIV_SUBSET,GSYM SET_EQ_SUBSET])

val eqf_eq = save_thm("eqf_eq",IMP_ANTISYM_RULE eqf_eq_lemma2 eqf_eq_lemma1)



(* ------------------------------------------------------------------------ *)
(* Equivalence and substitution over negation                                *)
(*                                                                            *)
(*                                                                            *)
(* ------------------------------------------------------------------------ *)
val eqf_notf =
TAC_PROOF
(([],
``!M Oi Os f f'.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (notf f) ==>
  (M,Oi,Os) sat (notf f')``),
```

```
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,notf_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_notf",eqf_notf)



(* ------------------------------------------------------------------------ *)
(* Equivalence and substitution over conjunction                            *)
(*                                                                          *)
(*                                                                          *)
(* ------------------------------------------------------------------------ *)
val eqf_andf1 =
TAC_PROOF
(([],
``!M Oi Os f f' g.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (f andf g) ==>
  (M,Oi,Os) sat (f' andf g)``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,andf_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_andf1",eqf_andf1)

val eqf_andf2 =
TAC_PROOF
(([],
``!M Oi Os f f' g.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (g andf f) ==>
  (M,Oi,Os) sat (g andf f')``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,andf_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_andf2",eqf_andf2)



(* ------------------------------------------------------------------------ *)
(* Equivalence and substitution over disjunction                            *)
(*                                                                          *)
(*                                                                          *)
(* ------------------------------------------------------------------------ *)
val eqf_orf1 =
TAC_PROOF
(([],
``!M Oi Os f f' g.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (f orf g) ==>
  (M,Oi,Os) sat (f' orf g)``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,orf_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_orf1",eqf_orf1)

val eqf_orf2 =
TAC_PROOF
(([],
``!M Oi Os f f' g.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (g orf f) ==>
  (M,Oi,Os) sat (g orf f')``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,orf_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_orf2",eqf_orf2)
```

```
(* ---------------------------------------------------------------------------- *)
(* Equivalence and substitution over implication                                *)
(*                                                                              *)
(*                                                                              *)
(* ---------------------------------------------------------------------------- *)
val eqf_impf1 =
TAC_PROOF
(([],
``!M Oi Os f f' g.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (f impf g) ==>
  (M,Oi,Os) sat (f' impf g)``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,impf_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_impf1",eqf_impf1)

val eqf_impf2 =
TAC_PROOF
(([],
``!M Oi Os f f' g.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (g impf f) ==>
  (M,Oi,Os) sat (g impf f')``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,impf_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_impf2",eqf_impf2)



(* ---------------------------------------------------------------------------- *)
(* Equivalence and substitution over equivalence                                *)
(*                                                                              *)
(*                                                                              *)
(* ---------------------------------------------------------------------------- *)
val eqf_eqf1 =
TAC_PROOF
(([],
``!M Oi Os f f' g.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (f eqf g) ==>
  (M,Oi,Os) sat (f' eqf g)``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,eqf_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_eqf1",eqf_eqf1)

val eqf_eqf2 =
TAC_PROOF
(([],
``!M Oi Os f f' g.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (g eqf f) ==>
  (M,Oi,Os) sat (g eqf f')``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,eqf_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_eqf2",eqf_eqf2)

(* ---------------------------------------------------------------------------- *)
(* Equivalence and substitution over says                                       *)
(*                                                                              *)
(*                                                                              *)
(* ---------------------------------------------------------------------------- *)
val eqf_says =
```

```
TAC_PROOF
(([],
``!M Oi Os P f f'.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (P says f) ==>
  (M,Oi,Os) sat (P says f')``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,says_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_says",eqf_says)


(* -------------------------------------------------------------------------- *)
(* Equivalence and substituion over controls                                  *)
(*                                                                            *)
(*                                                                            *)
(* -------------------------------------------------------------------------- *)
val eqf_controls =
TAC_PROOF
(([],
``!M Oi Os P f f'.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (P controls f) ==>
  (M,Oi,Os) sat (P controls f')``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,controls_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_controls",eqf_controls)


(* -------------------------------------------------------------------------- *)
(* Equivalence and substitution over reps                                     *)
(*                                                                            *)
(*                                                                            *)
(* -------------------------------------------------------------------------- *)
val eqf_reps =
TAC_PROOF
(([],
``!M Oi Os P Q f f'.
  ((M:('a,'b,'c,'d,'e)Kripke),Oi:'d po,Os:'e po) sat (f eqf f') ==> (M,Oi,Os) sat (reps P Q f) ==>
  (M,Oi,Os) sat (reps P Q f')``),
REPEAT GEN_TAC THEN
REWRITE_TAC[sat_def,reps_def,eqf_eq] THEN
DISCH_THEN (fn th => REWRITE_TAC[th]))

val _ = save_thm("eqf_reps",eqf_reps)

(**********
* Property of cmp
**********)

val Image_cmp =
store_thm
  ("Image_cmp",
   ``!R1 R2 R3:'k->bool u:'g.(((R1 O R2) u) SUBSET R3) =
     ((R2 u) SUBSET {y:'h | (R1 y) SUBSET R3})``,
  REPEAT GEN_TAC THEN
  REWRITE_TAC [SUBSET_DEF] THEN
  CONV_TAC (LAND_CONV (REWRITE_CONV [SPECIFICATION])) THEN
  CONV_TAC (DEPTH_CONV SET_SPEC_CONV) THEN
  REWRITE_TAC [O_DEF, SPECIFICATION] THEN
  EQ_TAC THENL
  [STRIP_GOAL_THEN (fn axi => REPEAT STRIP_TAC THEN MATCH_MP_TAC axi)
   THEN EXISTS_TAC (Term`x:'h`) THEN ASM_REWRITE_TAC []
  ,REPEAT STRIP_TAC THEN RES_TAC
  ]);
```

```
val Image_lemma =
TAC_PROOF(
  ([],
  ``~(!(x' :'b).
            Jext (jKS (M :('a, 'b, 'c, 'd, 'e) Kripke)) (P :'c Princ) x
              x' ==>
            !(x :'b).
              Jext (jKS M) (Q :'c Princ) x' x ==>
              Efn (Oi :'d po) (Os :'e po) M (f :('a, 'c, 'd, 'e) Form) x)
        \/
        !(x' :'b).
          Jext (jKS M) P x x' ==>
          !(x :'b). Jext (jKS M) Q x' x ==> Efn Oi Os M f x``),
 PROVE_TAC []);

val Quoting =
store_thm
   ("Quoting",
   ``!M Oi Os P Q f.(M,Oi,Os) sat ((P quoting Q) says f) eqf
                                                  (P says (Q says f))``,
   REPEAT GEN_TAC THEN
   REWRITE_TAC
   [eqf_and_impf, sat_def, andf_def, INTER_DEF, Efn_def,
    quoting_def, Image_cmp] THEN
   REWRITE_TAC [DIFF_DEF, IN_UNIV, SUBSET_DEF, UNION_DEF] THEN
   CONV_TAC (DEPTH_CONV SET_SPEC_CONV) THEN
   REWRITE_TAC [IN_DEF] THEN
   CONV_TAC(DEPTH_CONV BETA_CONV) THEN
   REWRITE_TAC [Image_lemma, GSPEC_T]);


(*****************************
* Quoting_Eq
*
* P quoting Q says f = P says Q says f
*****************************)
val Quoting_Eq =
store_thm
   ("Quoting_Eq",
   ``!M Oi Os P Q f. ((M,Oi,Os) sat P quoting Q says f) =
                     ((M,Oi,Os) sat P says Q says f)``,
   REPEAT GEN_TAC THEN
   REWRITE_TAC [sat_def, Efn_def, quoting_def, Image_cmp]);

(***************************
* Controls_Eq
*
* P controls f = (P says f) impf f
*
****************************)

val Controls_Eq =
store_thm
  ("Controls_Eq",
  ``!M Oi Os P f. (M,Oi,Os) sat (P controls f) =
                  (M,Oi,Os) sat ((P says f) impf f)``,
  PROVE_TAC [sat_def, controls_says]);

(***************************
*Reps_Eq
*
* reps P Q f = (P quoting Q) says f impf (Q says f)
*
****************************)
val reps_def_lemma =
store_thm
   ("reps_def_lemma",
```

```
   ``!M Oi Os P Q f. Efn Oi Os M (reps P Q f) =
       Efn Oi Os M ((P quoting Q) says f impf (Q says f))``,
   REWRITE_TAC [reps_def, says_def, impf_def]);

val Reps_Eq =
store_thm
   ("Reps_Eq",
    ``!M Oi Os P Q f. (M,Oi,Os) sat (reps P Q f) =
           (M,Oi,Os) sat ((P quoting Q) says f impf (Q says f))``,
    PROVE_TAC [sat_def, reps_def_lemma]);

(***************************
*
* And_Says_Eq
*
****************************)
val And_Says_Eq =
let
 val th1 =
 SPECL
 [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``, ``(Oi :'d po)``, ``(Os :'e po)``,
  ``(P meet Q says f):('a,'c,'d,'e)Form``,``(P says f andf Q says f):('a,'c,'d,'e)Form``] eqf_sat;
 val th2 = SPEC_ALL And_Says
in
 MP th1 th2
end;

val _ = save_thm("And_Says_Eq",And_Says_Eq);



(* -------------------------------------------------------------------------- *)
(* (M,Oi,Os) sat TT                                                            *)
(* -------------------------------------------------------------------------- *)
val sat_TT =
TAC_PROOF(([],``(M,Oi,Os) sat TT``),
REWRITE_TAC[sat_def,TT_def])

val _ = save_thm("sat_TT",sat_TT)

val _ = print_theory "-";
val _ = export_theory ();

end;
```

D.4 aclDrules Theory

```
(* Created by SKC 2/19/2009. *)
(* These are derived rules. *)

(* Interactive mode
set_trace "Unicode" 0;
app load ["pred_setTheory", "pred_setLib", "relationTheory", "aclfoundationTheory",
          "aclsemanticsTheory", "aclrulesTheory"];
*)

structure aclDrulesScript = struct

open HolKernel boolLib Parse;
open bossLib
open pred_setLib pred_setTheory relationTheory;
open aclfoundationTheory aclsemanticsTheory aclrulesTheory;

val _ = new_theory "aclDrules";
```

```
(***************************
* Simplification1             Simplification2
*
*   f1  andf  f2                    f1  andf  f2
*  ____________                    ____________
*         f1                                f2
***************************)
(*****Proof of INTER_EQ_UNIV thanks to Lockwood Morris***********)
(*       It's much shorter than my original version below!                    *)
val INTER_EQ_UNIV =
store_thm
  ("INTER_EQ_UNIV",
   Term `!s1 s2:'a -> bool. (s1 INTER s2 = UNIV) = (s1 = UNIV) /\ (s2 = UNIV)`,
   REPEAT GEN_TAC
   THEN REWRITE_TAC [EXTENSION, IN_INTER, IN_UNIV]
   THEN CONV_TAC (RAND_CONV AND_FORALL_CONV) THEN REFL_TAC);

(********** Old proof of INTER_EQ_UNIV
val INTER_EQ_UNIV_lemma1 =
TAC_PROOF(
   ([], ``!s1 s2.(s1 INTER s2 = UNIV) ==> ((s1 = UNIV) /\ (s2 = UNIV))``),
  REPEAT GEN_TAC THEN
  REWRITE_TAC [UNIV_DEF, INTER_DEF,IN_DEF,(REWRITE_RULE [SPECIFICATION] EXTENSION)] THEN
  CONV_TAC (DEPTH_CONV BETA_CONV) THEN
  REWRITE_TAC [SYM(SPEC_ALL FORALL_AND_THM)] THEN
  DISCH_TAC THEN
  GEN_TAC THEN
  UNDISCH_TAC ``!(x :'a). {x | (s1 :'a -> bool) x /\ (s2 :'a -> bool) x} x`` THEN
  DISCH_THEN
  (fn th =>
        MP_TAC
        (SUBS
         [SYM(SPECL [``{(x :'a) | (s1 :'a -> bool) x /\ (s2 :'a -> bool) x}``, ``x``] SPECIFICATION)]
         (SPEC_ALL th))) THEN
  REWRITE_TAC [SET_SPEC_CONV ``(x :'a) IN {x | (s1 :'a -> bool) x /\ (s2 :'a -> bool) x}``]);

val INTER_EQ_UNIV_lemma2 =
TAC_PROOF(
  ([], ``!s1 s2.((s1 = UNIV) /\ (s2 = UNIV)) ==> (s1 INTER s2 = UNIV)``),
  REPEAT STRIP_TAC THEN
  ASM_REWRITE_TAC [INTER_UNIV]);

val INTER_EQ_UNIV =
store_thm
 ("INTER_EQ_UNIV",``!s1 s2.(s1 INTER s2 = UNIV) = ((s1 = UNIV) /\ (s2 = UNIV))``,
  REPEAT GEN_TAC THEN
  EQ_TAC THEN
  REWRITE_TAC [INTER_EQ_UNIV_lemma1, INTER_EQ_UNIV_lemma2]);
***************old proof of INTER_EQ_UNIV************)

val Simplification1 =
store_thm
   ("Simplification1",
    ``!M Oi Os f1 f2. ((M,Oi,Os) sat (f1 andf f2)) ==> ((M,Oi,Os) sat f1)``,
  REPEAT GEN_TAC THEN
  PROVE_TAC [sat_def, Efn_def, INTER_EQ_UNIV]);

val Simplification2 =
store_thm
   ("Simplification2",
    ``!M Oi Os f1 f2. ((M,Oi,Os) sat (f1 andf f2)) ==> ((M,Oi,Os) sat f2)``,
  REPEAT GEN_TAC THEN
  PROVE_TAC [sat_def, Efn_def, INTER_EQ_UNIV]);


(********ACL INFERENCE RULES FOR HOL*****************************)
(* These reduce the need for dealing with (M,Oi,Os) sat                    *)
```

```
(*****************************************************************)
val ACL_TAUT_TAC =
    REWRITE_TAC
    [sat_allworld, world_T, world_F, world_not,
     world_and, world_or, world_imp, world_eq,
     world_eqn, world_lte, world_lt]
    THEN DECIDE_TAC;
fun ACL_TAUT f =
    TAC_PROOF(([],(Term `(M,Oi,Os) sat ^f`)),
    ACL_TAUT_TAC);
fun ACL_ASSUM f = ASSUME (Term `(M,Oi,Os) sat ^f`);
fun ACL_MP th1 th2 = MATCH_MP (MATCH_MP (SPEC_ALL Modus_Ponens) th1) th2;
fun ACL_SIMP1 th = MATCH_MP Simplification1 th;
fun ACL_SIMP2 th = MATCH_MP Simplification2 th;


(***************************
* Controls
*
* P controls f     P says f
*--------------------------
*              f
***************************)
val Controls =
let
        val a1 = ACL_ASSUM ``(P:'c Princ) controls (f:('a,'c,'d,'e)Form)``
        val a2 = ACL_ASSUM ``(P:'c Princ) says (f:('a,'c,'d,'e)Form)``
        val th3 = REWRITE_RULE [Controls_Eq] a1
        val th4 = ACL_MP a2 th3
        val th5 =
         GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                ``(P :'c Princ)``,``(f :('a, 'c, 'd, 'e) Form)``]
                (DISCH_ALL th4)
in
        save_thm("Controls",th5)
end;


(*******ACL INFERENCE RULES FOR HOL*****************************)
(* These reduce the need for dealing with (M,Oi,Os) sat            *)
(*****************************************************************)
fun CONTROLS th1 th2 = MATCH_MP (MATCH_MP Controls th2) th1;

(***************************
* Reps
*
*  Q controls f    reps P Q f    P quoting Q says f
*  ------------------------------------------------
*                        f
***************************)
val Reps =
let
        val a1 = ACL_ASSUM ``(Q:'c Princ) controls (f:('a,'c,'d,'e)Form) ``
        val a2 = ACL_ASSUM ``reps (P:'c Princ) (Q:'c Princ) (f:('a,'c,'d,'e)Form)``
        val a3 = ACL_ASSUM ``((P:'c Princ) quoting (Q:'c Princ)) says (f:('a,'c,'d,'e)Form)``
        val th4 = REWRITE_RULE [Reps_Eq] a2
        val th5 = ACL_MP a3 th4
        val th6 = REWRITE_RULE [Controls_Eq] a1
        val th7 = ACL_MP th5 th6
        val th8 = GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                        ``(P :'c Princ)``,``(Q: 'c Princ)``,``(f :('a, 'c, 'd, 'e) Form)``]
                        (DISCH_ALL th7)
in
        save_thm("Reps",th8)
end;

(***************************
```

```
* Rep_Controls_eq
*
*------------------------------------------------
* reps A B f = A controls (B says f)
***************************)

val Rep_Controls_Eq =
store_thm
   ("Rep_Controls_Eq",
    ``!M Oi Os (A:'c Princ) B (f:('a,'c,'d,'e)Form).
      ((M,Oi,Os) sat reps A B f) = ((M,Oi,Os) sat (A controls (B says f)))``,
   REWRITE_TAC [Reps_Eq, Controls_Eq, sat_def,Efn_def, quoting_def, Image_cmp]);

(***************************
* Rep_Says
*
* rep A B f     A quoting B says f
*-------------------------------------------
*               B says f
***************************)
val Rep_Says =
let
        val a1 = ACL_ASSUM ``reps (P:'c Princ) (Q:'c Princ) (f:('a,'c,'d,'e)Form)``
        val a2 = ACL_ASSUM ``(P:'c Princ) quoting (Q:'c Princ) says (f:('a,'c,'d,'e)Form)``
        val th3 = REWRITE_RULE [Rep_Controls_Eq] a1
        val th4 = REWRITE_RULE [Quoting_Eq] a2
        val th5 = CONTROLS th3 th4
        val th6 = GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                        ``(P :'c Princ)``,``(Q: 'c Princ)``,``(f :('a, 'c, 'd, 'e) Form)``]
                        (DISCH (hd (hyp a1)) (DISCH (hd (hyp a2)) th5))
in
        save_thm("Rep_Says",th6)
end;

(*************************************
* Conjunction
*
*     f1    f2
*   -----------
*   f1 andf f2
*************************************)
val Conjunction =
store_thm
   ("Conjunction",
    ``!M Oi Os f1 f2.(M,Oi,Os) sat f1 ==> (M,Oi,Os) sat f2 ==> (M,Oi,Os) sat (f1 andf f2)``,
  REPEAT GEN_TAC THEN
  REWRITE_TAC [sat_def,world_and,andf_def] THEN
  REPEAT DISCH_TAC THEN
  ASM_REWRITE_TAC [INTER_EQ_UNIV]);

(*************************************
* Disjunction1
*
*       f1
*   -----------
*   f1 orf f2
*************************************)
val Disjunction1 =
store_thm
  ("Disjunction1",
   ``!M Oi Os f1 f2.(M,Oi,Os) sat f1 ==> (M,Oi,Os) sat f1 orf f2``,
  REPEAT GEN_TAC THEN
  REWRITE_TAC [sat_def,world_or,orf_def] THEN
  REPEAT DISCH_TAC THEN
  ASM_REWRITE_TAC [UNION_UNIV]);

(*************************************
```

```
* Disjunction2
*
*       f2
*  ----------
*  f1 orf f2
*************************************)
val Disjunction2 =
store_thm
  ("Disjunction2",
   ``!M Oi Os f1 f2.(M,Oi,Os) sat f2 ==> (M,Oi,Os) sat f1 orf f2``,
  REPEAT GEN_TAC THEN
  REWRITE_TAC [sat_def,world_or,orf_def] THEN
  REPEAT DISCH_TAC THEN
  ASM_REWRITE_TAC [UNION_UNIV]);

(*************************************
* Modus Tollens
*
*  f1 impf f2     notf f2
*  ---------------------
*            notf f1
*************************************)
val Modus_Tollens =
let
    val th1 = ACL_ASSUM ``(f1:('a,'c,'d,'e)Form) impf (f2:('a,'c,'d,'e)Form)``
    val th2 = ACL_ASSUM ``notf (f2:('a,'c,'d,'e)Form)``
    val th3 = ACL_TAUT ``(f1:('a,'c,'d,'e)Form) impf (f2:('a,'c,'d,'e)Form) eqf (notf f2 impf notf f1)``
    val th4 = REWRITE_RULE [eqf_and_impf] th3
    val th5 = ACL_SIMP1 th4
    val th6 = ACL_MP th1 th5
    val th7 = ACL_MP th2 th6
    val th8 = GENL
        [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
        ``(f1 :('a,'c, 'd,'e) Form)``,``(f2 :('a, 'c, 'd, 'e) Form)``]
        (DISCH_ALL th7)
in
   save_thm("Modus_Tollens",th8)
end;

(*************************************
* Double Negation
*
*    notf (notf f)
*  ---------------
*          f
*************************************)
val Double_Negation =
let
val th1 = ACL_ASSUM ``notf (notf (f:('a,'c,'d,'e)Form))``
val th2 = ACL_TAUT ``(notf(notf (f:('a,'c,'d,'e)Form))) eqf (f:('a,'c,'d,'e)Form)``
val th3 = MATCH_MP eqf_sat th2
val th4 = REWRITE_RULE [th3] th1
val th5 =
 GENL
 [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
  ``(f :('a, 'c, 'd, 'e) Form)``]
(DISCH_ALL th4)
in
 save_thm("Double_Negation",th5)
end;

(*************************************
* Hypothetical Syllogism
*
*   f1 impf f2     f2 impf f3
*  -------------------------
*           f1 impf f3
```

```
*************************************)

val Hypothetical_Syllogism =
let
 val th1 = ACL_ASSUM ``(f1:('a,'c,'d,'e)Form) impf (f2:('a,'c,'d,'e)Form)``
 val th2 = ACL_ASSUM ``(f2:('a,'c,'d,'e)Form) impf (f3:('a,'c,'d,'e)Form)``
 val th3 =
  ACL_TAUT
  ``((f1:('a,'c,'d,'e)Form) impf (f2:('a,'c,'d,'e)Form)) impf
    (f2 impf (f3:('a,'c,'d,'e)Form)) impf (f1 impf f3)``
 val th4 = ACL_MP th1 th3
 val th5 = ACL_MP th2 th4
 val th6 =
  DISCH ``((M :('a,'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
          (f2 :('a,'c, 'd, 'e) Form) impf (f3 :('a, 'c, 'd, 'e) Form)`` th5
 val th7 =
   GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
         ``(f1 :('a, 'c, 'd, 'e) Form)``,``(f2 :('a, 'c, 'd, 'e) Form)``,
         ``(f3 :('a, 'c, 'd, 'e) Form)``]
   (DISCH_ALL th6)
in
 save_thm("Hypothetical_Syllogism",th7)
end;

fun HS th1 th2 = MATCH_MP(MATCH_MP (SPEC_ALL Hypothetical_Syllogism) th1) th2;

(*************************************
* Disjunctive Syllogism
*
*   f1 orf f2   notf f1
*  -----------------------
*           f1
*************************************)
val Disjunctive_Syllogism =
let
        val th1 = ACL_ASSUM ``notf (f1:('a,'c,'d,'e)Form)``
        val th2 = ACL_ASSUM ``(f1:('a,'c,'d,'e)Form) orf f2``
        val th3 = ACL_TAUT ``((f1:('a,'c,'d,'e)Form) orf f2) impf (notf f1) impf f2``
        val th4 = ACL_MP th2 th3
        val th5 = ACL_MP th1 th4
        val th6 =
            GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                     ``(f1 :('a, 'c, 'd, 'e) Form)``,``(f2 :('a, 'c, 'd, 'e) Form)``]
                   (DISCH_ALL th5)
in
        save_thm("Disjunctive_Syllogism",th6)
end;

fun SAYS princ form =
    ISPECL [princ,form]
    (ISPECL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``]Says);

fun MP_SAYS princ f1 f2 =
    ISPECL [princ, f1, f2](SPECL [``M:('a,'b,'c,'d,'e)Kripke``, ``Oi:'d po``, ``Os: 'e po``] MP_Says);

(*************************************
* Says Simplification 1
*
*   P says (f1 andf f2)
*  --------------------
*        P says f1
*************************************)
val Says_Simplification1 =
let
        val th1 = ACL_ASSUM ``P says ((f1:('a,'c,'d,'e)Form) andf f2)``
        val th2 = ACL_TAUT ``((f1:('a,'c,'d,'e)Form) andf f2) impf f1``
        val th3 = SAYS ``(P :'c Princ)`` ``((f1:('a,'c,'d,'e)Form) andf f2) impf f1``
```

```
        val th4 = MP th3 th2
        val th5 =
            MP_SAYS ``(P:'c Princ)`` ``(f1:('a,'c,'d,'e)Form) andf f2`` ``(f1:('a,'c,'d,'e)Form)``
        val th6 = ACL_MP th4 th5
        val th7 = ACL_MP th1 th6
        val th8 =
            GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                  ``(P :'c Princ)``,``(f1 :('a, 'c, 'd, 'e) Form)``,``(f2 :('a, 'c, 'd, 'e) Form)``]
                  (DISCH_ALL th7)
in
        save_thm("Says_Simplification1",th8)
end;


(*************************************
* Says Simplification 2
*
*   P says (f1 andf f2)
*  -------------------
*        P says f2
*************************************)
val Says_Simplification2 =
let
        val th1 = ACL_ASSUM ``P says ((f1:('a,'c,'d,'e)Form) andf f2)``
        val th2 = ACL_TAUT ``((f1:('a,'c,'d,'e)Form) andf f2) impf f2``
        val th3 = SAYS ``(P :'c Princ)`` ``((f1:('a,'c,'d,'e)Form) andf f2) impf f2``
        val th4 = MP th3 th2
        val th5 =
            MP_SAYS ``(P:'c Princ)`` ``(f1:('a,'c,'d,'e)Form) andf f2`` ``(f2:('a,'c,'d,'e)Form)``
        val th6 = ACL_MP th4 th5
        val th7 = ACL_MP th1 th6
        val th8 = GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                  ``(P :'c Princ)``,``(f1 :('a, 'c, 'd, 'e) Form)``,``(f2 :('a, 'c, 'd, 'e) Form)``]
                  (DISCH_ALL th7)
in
        save_thm("Says_Simplification2",th8)
end;

(*************************************
* Derived Speaks For
*
*   P speaks_for Q   P says f
*  --------------------------
*             Q says f
*************************************)
val Derived_Speaks_For =
let
        val th1 = ACL_ASSUM ``(P speaks_for Q):('a,'c,'d,'e)Form``
        val th2 = ACL_ASSUM ``(P: 'c Princ) says (f:('a,'c,'d,'e)Form)``
        (* For some reason, need to eliminate all quantifiers of Speaks_For *)
        val th3 = ACL_MP th1 (SPEC_ALL Speaks_For)
        val th4 = ACL_MP th2 th3
        val th5 = GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                  ``(P :'c Princ)``,``(Q :'c Princ)``,``(f :('a, 'c, 'd, 'e) Form)``]
                  (DISCH_ALL th4)
in
        save_thm("Derived_Speaks_For",th5)
end;

(********************************************
* Derived Controls
*
*  P speaks_for Q    Q controls f
*  ------------------------------
*             P controls f
********************************************)
val Derived_Controls =
```

```
let
        val th1 = ACL_ASSUM ``(P speaks_for Q):('a,'c,'d,'e)Form``
        val th2 = ACL_ASSUM ``Q controls (f:('a,'c,'d,'e)Form)``
        val th3 = REWRITE_RULE [Controls_Eq] th2
        val th4 = ACL_MP th1 (SPEC_ALL Speaks_For)
        val th5 = HS th4 th3
        val th6 = REWRITE_RULE [SYM(SPEC_ALL Controls_Eq)] th5
        val th7 = GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                            ``(P :'c Princ)``,``(Q :'c Princ)``,``(f :('a, 'c, 'd, 'e) Form)``]
                           (DISCH_ALL th6)
in
        save_thm("Derived_Controls",th7)
end;

fun DC th1 th2 = MATCH_MP(MATCH_MP (SPEC_ALL Derived_Controls) th1) th2;


fun DOMS_TRANS th1 th2 = MATCH_MP(MATCH_MP (SPEC_ALL doms_transitive) th1) th2;
(****************************
* sl doms
*
*   sl(P) eqs l1    sl(Q) eqs l2    l2 doms l1
*  ------------------------------------------
*                sl(Q) doms sl(P)
****************************)

val sl_doms =
let
        val th1 = ACL_ASSUM ``(sl(P) eqs l1):('a,'c,'d,'e)Form``
        val th2 = ACL_ASSUM ``(sl(Q) eqs l2):('a,'c,'d,'e)Form``
        val th3 = ACL_ASSUM ``(l2 doms l1):('a,'c,'d,'e)Form``
        val th4 = REWRITE_RULE [eqs_Eq] th1
        val th5 = REWRITE_RULE [eqs_Eq] th2
        val th6 = ACL_SIMP1 th4
        val th7 = DOMS_TRANS th3 th6
        val th8 = ACL_SIMP2 th5
        val th9 = DOMS_TRANS th8 th7
        val th10 =
            DISCH
            ``(M,Oi,Os) sat sl P eqs l1``
            (DISCH ``(M,Oi,Os) sat sl Q eqs l2``
            (DISCH ``(M,Oi,Os) sat l2 doms l1`` th9))
        val th11 =
            GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                   ``(P:'c)``,``(Q:'c)``,``(l1 :('c, 'e) SecLevel)``,``(l2 :('c, 'e) SecLevel)``] th10
in
        save_thm("sl_doms",th11)
end;

fun SL_DOMS th1 th2 th3 = MATCH_MP(MATCH_MP(MATCH_MP sl_doms th1) th2) th3;
fun DOMI_TRANS th1 th2 = MATCH_MP(MATCH_MP (SPEC_ALL domi_transitive) th1) th2;
(****************************
* il domi
*
*   il(P) eqi l1    il(Q) eqi l2    l2 domi l1
*  ------------------------------------------
*                il(Q) domi il(P)
****************************)

val il_domi =
let
        val th1 = ACL_ASSUM ``(il(P) eqi l1):('a,'c,'d,'e)Form``
        val th2 = ACL_ASSUM ``(il(Q) eqi l2):('a,'c,'d,'e)Form``
        val th3 = ACL_ASSUM ``(l2 domi l1):('a,'c,'d,'e)Form``
        val th4 = REWRITE_RULE [eqi_Eq] th1
        val th5 = REWRITE_RULE [eqi_Eq] th2
        val th6 = ACL_SIMP1 th4
```

```
        val th7 = DOMI_TRANS th3 th6
        val th8 = ACL_SIMP2 th5
        val th9 = DOMI_TRANS th8 th7
        val th10 =
            DISCH
            ``(M,Oi,Os) sat il P eqi l1``
            (DISCH ``(M,Oi,Os) sat il Q eqi l2``
            (DISCH ``(M,Oi,Os) sat l2 domi l1`` th9))
        val th11 =
            GENL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,``(Os :'e po)``,
                  ``(P:'c)``,``(Q:'c)``,``(l1 :('c, 'd) IntLevel)``,``(l2 :('c, 'd) IntLevel)``] th10
in
        save_thm("il_domi",th11)
end;

(**********************************************************
* IL_DOMI
*
* IL_DOMI : thm -> thm -> thm -> thm
*
* SYNOPSIS
* Applies il_domi to theorems in the access-control logic
*
* DESCRIPTION
*
*  A1 |- (M,Oi,Os) sat il P eqi l1   A2 |- (M,Oi,Os) sat il Q eqi l2   A3 |- (M,Oi,Os) sat l2 domi l1
*  ------------------------------------------------------------------------------------------- IL_DOMI
*                                  A1 u A2 u A3 |- (M,Oi,Os) sat il Q domi il P
*
* FAILURE
* Fails unless the input theorems match in their corresponding terms in the
* access-control logic
**********************************************************)
fun IL_DOMI th1 th2 th3 = MATCH_MP(MATCH_MP(MATCH_MP il_domi th1) th2) th3;

val th1 = ACL_ASSUM ``(c1 eqn n1):('a,'c,'d,'e)Form``;
val th2 = ACL_ASSUM ``(c2 eqn n2):('a,'c,'d,'e)Form``;
val th3 = ACL_ASSUM ``(n1 lte n2):('a,'c,'d,'e)Form``;
val th4 = REWRITE_RULE[sat_def,eqn_def] th1;


(**********************************************
* val eqn_lte =
*     |- (M,Oi,Os) sat c1 eqn n1 ==>
*        (M,Oi,Os) sat c2 eqn n2 ==>
*        (M,Oi,Os) sat n1 lte n2 ==>
*        (M,Oi,Os) sat c1 lte c2 : thm
**********************************************)
val eqn_lte =
save_thm("eqn_lte",
TAC_PROOF(
 ([],
 ``((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
     (((c1 :num) eqn (n1 :num)) :('a, 'c, 'd, 'e) Form) ==>
   ((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((c2 :num) eqn (n2 :num)) :('a, 'c, 'd, 'e) Form) ==>
   ((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((n1 :num) lte (n2 :num)) :('a, 'c, 'd, 'e) Form) ==>
   ((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((c1 :num) lte (c2 :num)) :('a, 'c, 'd, 'e) Form)``),
(REWRITE_TAC[sat_def,eqn_def,lte_def] THEN
  COND_CASES_TAC THEN
  REWRITE_TAC[EMPTY_NOT_UNIV] THEN
  COND_CASES_TAC THEN
  REWRITE_TAC[EMPTY_NOT_UNIV] THEN
  ASM_REWRITE_TAC[])));
```

```
(*************************************************
* val eqn_lt =
*     |- (M,Oi,Os) sat c1 eqn n1 ==>
*        (M,Oi,Os) sat c2 eqn n2 ==>
*        (M,Oi,Os) sat n1 lt n2 ==>
*        (M,Oi,Os) sat c1 lt c2 : thm
*************************************************)
val eqn_lt =
save_thm("eqn_lt",
TAC_PROOF(
 ([],
 ``((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
    (((c1 :num) eqn (n1 :num)) :('a, 'c, 'd, 'e) Form) ==>
   ((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((c2 :num) eqn (n2 :num)) :('a, 'c, 'd, 'e) Form) ==>
   ((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((n1 :num) lt (n2 :num)) :('a, 'c, 'd, 'e) Form) ==>
   ((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((c1 :num) lt (c2 :num)) :('a, 'c, 'd, 'e) Form)``),
(REWRITE_TAC[sat_def,eqn_def,lt_def] THEN
  COND_CASES_TAC THEN
  REWRITE_TAC[EMPTY_NOT_UNIV] THEN
  COND_CASES_TAC THEN
  REWRITE_TAC[EMPTY_NOT_UNIV] THEN
  ASM_REWRITE_TAC[])));

(*************************************************
* val eqn_eqn =
*     |- (M,Oi,Os) sat c1 eqn n1 ==>
*        (M,Oi,Os) sat c2 eqn n2 ==>
*        (M,Oi,Os) sat n1 eqn n2 ==>
*        (M,Oi,Os) sat c1 eqn c2 : thm
*************************************************)
val eqn_eqn =
save_thm("eqn_eqn",
TAC_PROOF(
 ([],
 ``((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
    (((c1 :num) eqn (n1 :num)) :('a, 'c, 'd, 'e) Form) ==>
   ((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((c2 :num) eqn (n2 :num)) :('a, 'c, 'd, 'e) Form) ==>
   ((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((n1 :num) eqn (n2 :num)) :('a, 'c, 'd, 'e) Form) ==>
   ((M :('a, 'b, 'c, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
       (((c1 :num) eqn (c2 :num)) :('a, 'c, 'd, 'e) Form)``),
(REWRITE_TAC[sat_def,eqn_def,eqn_def] THEN
  COND_CASES_TAC THEN
  REWRITE_TAC[EMPTY_NOT_UNIV] THEN
  COND_CASES_TAC THEN
  REWRITE_TAC[EMPTY_NOT_UNIV] THEN
  ASM_REWRITE_TAC[])));

val _ = print_theory "-";
val _ = export_theory ();

end;
```

D.5 aclinfRules.sml

```
(* Created by S-K Chin 2/20/2009. modified by L.Morris 3/13/09 *)
(* These HOL/ml functions support the forward inference rule style of *)
(* reasoning in the access-control logic (see Access Control, Security,*)
(* and Trust: A Logical Approach, Shiu-Kai Chin and Susan Older, *)
(* CRC Press.*)
```

```
(* Modified by S-K Chin 11/9/2011. Added And_Says_LR and And_Says_RL *)

(* Modified by S. Perkins 8/12/2015. Added tactics section *)

structure acl_infRules :> acl_infRules =
struct

(* Interactive mode
set_trace "Unicode" 0;
app load ["pred_setTheory", "pred_setLib", "relationTheory", "aclfoundationTheory",
          "aclsemanticsTheory", "aclrulesTheory", "aclDrulesTheory",
          "pred_setSyntax","aclTermFuns"];
*)
(*********Load the theories on which the inference rules are based******)
open HolKernel boolLib Parse;
open bossLib pred_setLib pred_setTheory;
open aclfoundationTheory aclsemanticsTheory aclrulesTheory;
open aclDrulesTheory relationTheory;
open aclTermFuns pred_setSyntax;

(******* This tactic is from Lockwood Morris************)
(* modified by skc with the substitution of DECIDE_TAC   *)
(* for TAUT_TAC. DECIDE_TAC has superceded TAUT_TAC *)
(*********************************************************
* ACL_TAUT_TAC
*
* ACL_TAUT_TAC : tactic
*
* SYNOPSIS
* Invoke decision procedures to prove propositional formulas
* and partial order relations in the access-control logic.
*
* DESCRIPTION
* When given a propositional formula f in the access-control logic
* using only notf, andf, orf, impf, eqf,  eqn, lte, and lt,
* ACL_TAUT_TAC attempts to prove f true in all Kripke structures
* (M,Oi,Os).
*
*     A ?- (M,Oi,Os) sat f
*    =================== ACL_TAUT_TAC
*     A |-  (M,Oi,Os) sat f
*
* FAILURE
* Fails if f is not a propositional tautology, e.g., p and notf p.
*********************************************************)

val ACL_TAUT_TAC =
    REWRITE_TAC
    [sat_allworld, world_T, world_F, world_not,
     world_and, world_or, world_imp, world_eq,
     world_eqn, world_lte, world_lt]
    THEN DECIDE_TAC;

(*********************************************************
* ACL_TAUT
*
* ACL_TAUT : term -> thm
*
* SYNOPSIS
* Attempts to prove a proposition f in the access-control logic
* is true in all Kripke models (M,Oi,Os).
*
* DESCRIPTION
* When applied to a term f, which must have type Form,
* ACL_TAUT attempts to prove (M,Oi,Os) sat f.
*
```

```
*     ----------------  ACL_TAUT f
*     |- (M,Oi,Os) sat f
*
* FAILURE
* Fails if f is not a tautology.
***********************************************************)
(*******OLD DEFINITION***********
fun ACL_TAUT f =
    TAC_PROOF(([],(Term `(M,Oi,Os) sat ^f`)),
    ACL_TAUT_TAC);
******************************)
fun ACL_TAUT f =
let
 val f_type = type_of f
 val f_type_parts = dest_type f_type
 val [prop_type, name_type, integ_type, sec_type] = snd f_type_parts
 val M_type =
   mk_type ("Kripke",[prop_type, ``:'b``, name_type, integ_type, sec_type])
 val term =
   Term`((M : ^(ty_antiq M_type)),(Oi : ^(ty_antiq integ_type) po),
         (Os : ^(ty_antiq sec_type) po)) sat ^f`
 in
    TAC_PROOF(([],term),ACL_TAUT_TAC)
 end;
(***********************************************************
* ACL_ASSUM
*
* ACL_ASSUM : term -> thm
*
* SYNOPSIS
* Introduces an assumption in the access-control logic
*
* DESCRIPTION
* When applied to a term f, which must have type Form,
* ACL_ASSUM introduces a theorem
* (M,Oi,Os) sat f |- (M,Oi,Os) sat f.
*
*     ----------------------------  ACL_ASSUM f
*    (M,Oi,Os) sat f |- (M,Oi,Os) sat f
*
* FAILURE
* Fails unless f has type Form.
***********************************************************)
(*******OLD DEFINITION***********
fun ACL_ASSUM f = ASSUME
 (Term `((M:('a,'b,'c,'d,'e)Kripke),(Oi:'d po),(Os:'e po)) sat ^f`);
******************************)

fun ACL_ASSUM f =
let
 val f_type = type_of f
 val f_type_parts = dest_type f_type
 val [prop_type, name_type, integ_type, sec_type] = snd f_type_parts
 val M_type =
   mk_type ("Kripke",[prop_type, ``:'b``, name_type, integ_type, sec_type])
 val term =
   Term`((M : ^(ty_antiq M_type)),(Oi : ^(ty_antiq integ_type) po),
         (Os : ^(ty_antiq sec_type) po)) sat ^f`
 in
   ASSUME term
 end;

(***********************************************************
* ACL_ASSUM2
*
* ACL_ASSUM : term -> term -> term -> thm
*
```

```
* SYNOPSIS
* Introduces an assumption in the access-control logic
* given a formula f, and partial orderings on integrity
* labels Oi and security labels Os
*
* DESCRIPTION
* When applied to a term f, which must have type Form,
* Oi of type integ_type po, and Os of type sec_type po,
* ACL_ASSUMs introduces a theorem
* (M,Oi,Os) sat f |- (M,Oi,Os) sat f.
*
*     ----------------------------  ACL_ASSUM2 f Oi Os
*    (M,Oi,Os) sat f |- (M,Oi,Os) sat f
*
* FAILURE
* Fails unless f has type Form, and Oi and Os have types
* integ_type po and sec_type po, respectively
*********************************************************)
fun ACL_ASSUM2 f Oi Os =
let
 val f_type = type_of f
 val f_type_parts = dest_type f_type
 val [prop_type, name_type, integ_type, sec_type] = snd f_type_parts
 val M_type =
   mk_type ("Kripke",[prop_type, ``:'b``, name_type, integ_type, sec_type])
 val term =
   Term`((M : ^(ty_antiq M_type)),(^Oi : ^(ty_antiq integ_type) po),
         (^Os : ^(ty_antiq sec_type) po)) sat ^f`
 in
   ASSUME term
 end;

(*********************************************************
* ACL_MP
*
* ACL_MP : thm -> thm -> thm
*
* SYNOPSIS
* Implements Modus Ponens in the access-control logic
*
* DESCRIPTION
* When applied to theorems A1 |- (M,Oi,Os) sat f1 and
* A2 |- (M,Oi,Os) sat f1 impf f2 in the access-control logic,
* ACL_MP introduces a theorem A1 u A2 |- (M,Oi,Os) sat f2.
*
*     A1 |- (M,Oi,Os) sat f1   A2 |- (M,Oi,Os) sat f1 impf f2
*     -------------------------------------------------  ACL_MP
*                        A1 u A2 |- (M,Oi,Os) sat f2
*
* FAILURE
* Fails unless f1 in the first theorem is the same as f1 in the second
* theorem.
*********************************************************)
fun ACL_MP th1 th2 = MATCH_MP (MATCH_MP (SPEC_ALL Modus_Ponens) th1) th2;

(*********************************************************
* SAYS
*
* SAYS : term -> thm -> thm
*
* SYNOPSIS
* Applies the Says inference rule to a theorem A |- (M,Oi,Os) sat f
* in the access-control logic.
*
* DESCRIPTION
*
*             A |- (M,Oi,Os) sat f
```

```
*         ------------------------- SAYS P f
*         A |- (M,Oi,Os) sat P says f
*
* FAILURE
* Fails unless the input theorem is a double negation in the
* access-control logic
***********************************************************)
fun SAYS Q th = (SPEC Q (MATCH_MP Says th));

(******
fun SAYS princ form =
    ISPECL [princ,form]
    (ISPECL [``(M :('a, 'b, 'c, 'd, 'e) Kripke)``,``(Oi :'d po)``,
             ``(Os :'e po)``]Says);
*******)

(***********************************************************
* MP_SAYS
*
* MP_SAYS : term -> term -> term -> thm
*
* SYNOPSIS
* implements MP Says rule
*
* DESCRIPTION
*
*  -----------------------------------------------------  MP_SAYS P f1 f2
*  |- (M,Oi,Os) sat (P says (f1 impf f2)) impf
*     ((P says f1) impf (P says f2))
*
* FAILURE
* Fails unless princ is a principal,f1 and f2 are terms in the
* access-control logic, and princ, f1, and f2 have consistent
* types.
***********************************************************)
(*******OLD DEFINITION***********
fun MP_SAYS princ f1 f2 =
    ISPECL [princ, f1, f2](SPECL [``M:('a,'b,'c,'d,'e)Kripke``, ``Oi:'d po``, ``Os: 'e po``] MP_Says);
*******************************)
fun MP_SAYS princ f1 f2 =
let
 val f1_type = type_of f1
 val f1_type_parts = dest_type f1_type
 val [prop_type, name_type, integ_type, sec_type] = snd f1_type_parts
 val M_type =
   mk_type ("Kripke",[prop_type, ``:'b``, name_type, integ_type, sec_type])
 in
   ISPECL
   [``M : ^(ty_antiq M_type)``,``Oi : ^(ty_antiq integ_type) po``,
    ``Os : ^(ty_antiq sec_type) po``, princ, f1, f2]
   MP_Says
 end;

(***********************************************************
* ACL_MT
*
* ACL_MT : thm -> thm -> thm
*
* SYNOPSIS
* Implements Modus Tollens in the access-control logic
*
* DESCRIPTION
* When applied to theorems A1 |- (M,Oi,Os) sat notf f2 and
* A2 |- (M,Oi,Os) sat f1 impf f2 in the access-control logic,
* ACL_MT introduces a theorem A1 u A2 |- (M,Oi,Os) sat notf f1.
*
*     A1 |- (M,Oi,Os) sat f1 impf f2    A2 |- (M,Oi,Os) sat notf f2
```

```
*     -------------------------------------------------------  ACL_MT
*                         A1 u A2 |- (M,Oi,Os) sat notf f1
*
* FAILURE
* Fails unless f2 in the first theorem is the same as f2 in the second
* theorem.
**********************************************************)
fun ACL_MT th1 th2 = MATCH_MP (MATCH_MP (SPEC_ALL Modus_Tollens) th1) th2;

(**********************************************************
* ACL_SIMP1
*
* ACL_SIMP1 : thm -> thm
*
* SYNOPSIS
* Extracts left conjunct of a theorem in the access-control logic.
*
* DESCRIPTION
*
*     A |- (M,Oi,Os) sat f1 andf f2
*     --------------------------  ACL_SIMP1
*          A |- (M,Oi,Os) sat f1
*
* FAILURE
* Fails unless the input theorem is a conjunction in the
* access-control logic.
**********************************************************)
fun ACL_SIMP1 th = MATCH_MP (SPEC_ALL Simplification1) th;

(**********************************************************
* ACL_SIMP2
*
* ACL_SIMP2 : thm -> thm
*
* SYNOPSIS
* Extracts right conjunct of a theorem in the access-control logic.
*
* DESCRIPTION
*
*     A |- (M,Oi,Os) sat f1 andf f2
*     --------------------------  ACL_SIMP2
*          A |- (M,Oi,Os) sat f2
*
* FAILURE
* Fails unless the input theorem is a conjunction in the
* access-control logic.
**********************************************************)
fun ACL_SIMP2 th = MATCH_MP (SPEC_ALL Simplification2) th;

(**********************************************************
* ACL_CONJ
*
* ACL_CONJ : thm -> thm -> thm
*
* SYNOPSIS
* Introduces a conjunction in the access-control logic
*
* DESCRIPTION
*
*     A1 |- (M,Oi,Os) sat f1   A2 |- (M,Oi,Os) sat f2
*     ------------------------------------------------  ACL_CONJ
*                         A1 u A2 |- (M,Oi,Os) sat f1 andf f2
*
* FAILURE
* Fails unless both theorems are of the form A |- (M,Oi,Os) sat f.
**********************************************************)
fun ACL_CONJ th1 th2 = MATCH_MP(MATCH_MP (SPEC_ALL Conjunction) th1) th2;
```

```
(*********************************************************
* ACL_DISJ1
*
* ACL_DISJ1 : term -> thm -> thm
*
* SYNOPSIS
* Introduces a right disjunct into the conclusion of an access-control
* logic theorem
*
* DESCRIPTION
*
*               A |- (M,Oi,Os) sat f1
*         --------------------------  ACL_DISJ1 f2
*          A |- (M,Oi,Os) sat f1 orf f2
*
* FAILURE
* Fails unless the input theorem is a disjunction in the
* access-control logic and the types of f1 and f2 are the same.
*********************************************************)
(****** old definition *****
fun ACL_DISJ1 f th = (SPEC f) (GEN ``f2`` (MATCH_MP (SPEC_ALL Disjunction1) th));
*****)
fun ACL_DISJ1 f th =
let
 val f_type = type_of f
 val term = Term`f2:^(ty_antiq f_type)`
in
 SPEC f (GEN term (MATCH_MP (SPEC_ALL Disjunction1) th))
end;


(*********************************************************
* ACL_DISJ2
*
* ACL_DISJ2 : term -> thm -> thm
*
* SYNOPSIS
* Introduces a left disjunct into the conclusion of an access-control
* logic theorem
*
* DESCRIPTION
*
*               A |- (M,Oi,Os) sat f2
*         --------------------------  ACL_DISJ2 f1
*          A |- (M,Oi,Os) sat f1 orf f2
*
* FAILURE
* Fails unless the input theorem is a disjunction in the
* access-control logic and the types of f1 and f2 are the same.
*********************************************************)
(*******OLD DEFINITION*************
fun ACL_DISJ2 f1 th = (SPEC f1) (MATCH_MP (SPEC_ALL Disjunction2) th);
********************************)
fun ACL_DISJ2 f th =
let
 val f_type = type_of f
 val term = Term`f1:^(ty_antiq f_type)`
in
 SPEC f (GEN term (MATCH_MP (SPEC_ALL Disjunction2) th))
end;


(*********************************************************
* CONTROLS
*
```

```
* CONTROLS : thm->thm -> thm
*
* SYNOPSIS
* Deduces formula f if the principal who says f also controls f.
*
* DESCRIPTION
*
*     A1 |- (M,Oi,Os) sat P controls f   A2 |- (M,Oi,Os) sat P says f
*     --------------------------------------------------------  CONTROLS
*                         A1 u A2 |- (M,Oi,Os) sat f
*
* FAILURE
* Fails unless the theorems match in terms of principals and formulas
* in the access-control logic.
***********************************************************)
fun CONTROLS th1 th2 = MATCH_MP (MATCH_MP (SPEC_ALL Controls) th2) th1;

(************************************************************
* REPS
*
* REPS : thm -> thm -> thm -> thm
*
* SYNOPSIS
* Concludes statement f given theorems on delegation, quoting, and
* jurisdiction.
*
* DESCRIPTION
*
* A1 |- (M,Oi,Os) sat reps P Q f  A2 |- (M,Oi,Os) sat (P quoting Q) says f
*                            A3 |- (M,Oi,Os) sat Q controls f
*     --------------------------------------------------------  REPS
*                             A1 u A2 u A3 |- (M,Oi,Os) sat f
*
* FAILURE
* Fails unless M, Oi, Os, P, Q, and f match in all three theorems.
***********************************************************)
fun REPS th1 th2 th3 =
    MATCH_MP(MATCH_MP (MATCH_MP (SPEC_ALL Reps) th1) th2) th3;

(************************************************************
* REP_SAYS
*
* REP_SAYS : thm -> thm -> thm
*
* SYNOPSIS
* Concludes statement f given theorems on delegation, quoting, and
* jurisdiction.
*
* DESCRIPTION
*
* A1 |- (M,Oi,Os) sat reps P Q f  A2 |- (M,Oi,Os) sat (P quoting Q) says f
*     --------------------------------------------------------  REP_SAYS
*                             A1 u A2 |- (M,Oi,Os) sat Q says f
*
* FAILURE
* Fails unless M, Oi, Os, P, Q, and f match in all three theorems.
***********************************************************)
fun REP_SAYS th1 th2 = MATCH_MP (MATCH_MP (SPEC_ALL Rep_Says) th1) th2;

(************************************************************
* ACL_DN
*
* ACL_DN : thm -> thm
*
* SYNOPSIS
* Applies double negation to formula in the access-control logic
*
```

```
* DESCRIPTION
*
*         A |- (M,Oi,Os) sat notf(notf f)
*         ---------------------------  ACL_DN
*               A |- (M,Oi,Os) sat f
*
* FAILURE
* Fails unless the input theorem is a double negation in the
* access-control logic
*********************************************************)
fun ACL_DN th = MATCH_MP (SPEC_ALL Double_Negation) th;

(*********************************************************
* SPEAKS_FOR
*
* SPEAKS_FOR : thm -> thm -> thm
*
* SYNOPSIS
* Applies Derived Speaks For to theorems in the access-control logic
*
* DESCRIPTION
*
*  A1 |- (M,Oi,Os) sat P speaks_for Q   A2 |- (M,Oi,Os) sat P says f
*  ------------------------------------------------------------ SPEAKS_FOR
*                    A1 u A2 |- (M,Oi,Os) sat Q says f
*
* FAILURE
* Fails unless the first theorem is of the form P speaksfor Q, the
* second is P says f, and the types are the same.
*********************************************************)
fun SPEAKS_FOR th1 th2 =
             MATCH_MP (MATCH_MP (SPEC_ALL Derived_Speaks_For) th1) th2;


(*********************************************************
* HS
*
* HS : thm -> thm -> thm
*
* SYNOPSIS
* Applies hypothetical syllogism to theorems in the access-control logic
*
* DESCRIPTION
*
*  A1 |- (M,Oi,Os) sat f1 impf f2   A2 |- (M,Oi,Os) sat f2 impf f3
*  --------------------------------------------------------- HS
*               A1 u A2 |- (M,Oi,Os) sat f1 impf f3
*
* FAILURE
* Fails unless the input theorems match in their consequent and
* antecedent in access-control logic
*********************************************************)
fun HS th1 th2 = MATCH_MP(MATCH_MP (SPEC_ALL Hypothetical_Syllogism) th1) th2;


(*********************************************************
* DC
*
* DC : thm -> thm -> thm
*
* SYNOPSIS
* Applies Derived Controls rule to theorems in the access-control logic
*
* DESCRIPTION
*
*  A1 |- (M,Oi,Os) sat P speaks_for Q   A2 |- (M,Oi,Os) sat Q controls f
*  --------------------------------------------------------------- DC
```

```
*               A1 u A2 |- (M,Oi,Os) sat P controls f
*
* FAILURE
* Fails unless the input theorems match in their corresponding principal
* names
*********************************************************)
fun DC th1 th2 = MATCH_MP(MATCH_MP (SPEC_ALL Derived_Controls) th1) th2;

(*********************************************************
* SAYS_SIMP1
*
* SAYS_SIMP1 : thm -> thm
*
* SYNOPSIS
* Applies the Says_Simplification1 rule to conjunctive statements within
* says statements in theorems in the access-control logic
*
* DESCRIPTION
*
*   A |- (M,Oi,Os) sat P says (f1 andf f2)
*  ---------------------------------- SAYS_SIMP1
*          A |- (M,Oi,Os) sat P says f1
*
* FAILURE
* Fails unless the input theorem is a conjunction within a
* says statement in the access-control logic
*********************************************************)
fun SAYS_SIMP1 th = MATCH_MP (SPEC_ALL Says_Simplification1) th;

(*********************************************************
* SAYS_SIMP2
*
* SAYS_SIMP2 : thm -> thm
*
* SYNOPSIS
* Applies the Says_Simplification2 rule to conjunctive statements within
* says statements in theorems in the access-control logic
*
* DESCRIPTION
*
*   A |- (M,Oi,Os) sat P says (f1 andf f2)
*  ---------------------------------- SAYS_SIMP2
*          A |- (M,Oi,Os) sat P says f2
*
* FAILURE
* Fails unless the input theorem is a conjunction within a
* says statement in the access-control logic
*********************************************************)
fun SAYS_SIMP2 th = MATCH_MP (SPEC_ALL Says_Simplification2) th;

(*********************************************************
* DOMS_TRANS
*
* DOMS_TRANS : thm -> thm -> thm
*
* SYNOPSIS
* Applies transitivity of doms to theorems in the access-control logic
*
* DESCRIPTION
*
*  A1 |- (M,Oi,Os) sat l1 doms l2   A2 |- (M,Oi,Os) sat l2 doms l3
*  ------------------------------------------------------- DOMS_TRANS
*               A1 u A2 |- (M,Oi,Os) sat l1 doms l3
*
* FAILURE
* Fails unless l1, l2, and l3 match appropriately and have the
* same type.
```

```
*********************************************************)
fun DOMS_TRANS th1 th2 =
                MATCH_MP(MATCH_MP (SPEC_ALL doms_transitive) th1) th2;

(**********************************************************
* DOMI_TRANS
*
* DOMI_TRANS : thm -> thm -> thm
*
* SYNOPSIS
* Applies transitivity of domi to theorems in the access-control logic
*
* DESCRIPTION
*
*  A1 |- (M,Oi,Os) sat l1 domi l2   A2 |- (M,Oi,Os) sat l2 domi l3
*  ------------------------------------------------------- DOMI_TRANS
*               A1 u A2 |- (M,Oi,Os) sat l1 domi l3
*
* FAILURE
* Fails unless the input theorems match in their corresponding terms
*********************************************************)
fun DOMI_TRANS th1 th2 =
             MATCH_MP(MATCH_MP (SPEC_ALL domi_transitive) th1) th2;

(**********************************************************
* SL_DOMS
*
* SL_DOMS : thm -> thm -> thm -> thm
*
* SYNOPSIS
* Applies sl_doms to theorems in the access-control logic
*
* DESCRIPTION
*
*  A1 |- (M,Oi,Os) sat sl P eqs l1
*  A2 |- (M,Oi,Os) sat sl Q eqs l2
*  A3 |- (M,Oi,Os) sat l2 doms l1
*  --------------------------------------------------------------- SL_DOMS
*  A1 u A2 u A3 |- (M,Oi,Os) sat sl Q doms sl P
*
* FAILURE
* Fails unless the types are consistent across the three
* input theorems
*********************************************************)
fun SL_DOMS th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP sl_doms th1) th2) th3;

(**********************************************************
* IL_DOMI
*
* IL_DOMI : thm -> thm -> thm -> thm
*
* SYNOPSIS
* Applies il_doms to theorems in the access-control logic
*
* DESCRIPTION
*
*         A1 |- (M,Oi,Os) sat il P eqi l1
*         A2 |- (M,Oi,Os) sat il Q eqi l2
*         A3 |- (M,Oi,Os) sat l2 domi l1
*  -------------------------------------------- IL_DOMI
*  A1 u A2 u A3 |- (M,Oi,Os) sat il Q domi il P
*
* FAILURE
* Fails unless the types are consistent among the three
* theorems.
*********************************************************)
```

```
fun IL_DOMI th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP il_domi th1) th2) th3;

(**********************************************************
* QUOTING_RL
*
* QUOTING_RL : thm -> thm
*
* SYNOPSIS
* Applies quoting rule to theorems in the access-control logic
*
* DESCRIPTION
*    th [P says Q says f/A]
*  ------------------------- QUOTING_RL
*  th [P quoting Q says f/A]
*
* FAILURE
* Fails unless the input theorem is of the form P says Q
* says f.
***********************************************************)
fun QUOTING_RL th = REWRITE_RULE [GSYM(SPEC_ALL Quoting_Eq)] th;

(**********************************************************
* QUOTING_LR
*
* QUOTING_LR : thm -> thm
*
* SYNOPSIS
* Applies quoting rule to theorems in the access-control logic
*
* DESCRIPTION
*              th [P quoting Q says f/A]
*  -----------------------------QUOTING_LR
*              th [P says Q says f/A]
*
* FAILURE
* Fails unless the input theorem is of the form P quoting Q
* says f.
***********************************************************)
fun QUOTING_LR th = REWRITE_RULE [SPEC_ALL Quoting_Eq] th;

(**********************************************************
* EQN_LTE
*
* EQN_LTE : thm -> thm -> thm -> thm
*
* SYNOPSIS
* Applies eqn_lte to theorems in the access-control logic
*
* DESCRIPTION
*
*         A1 |- (M,Oi,Os) sat c1 eqn n1
*         A2 |- (M,Oi,Os) sat c2 eqn n2
*         A3 |- (M,Oi,Os) sat n1 lte n2
*  -------------------------------------------- EQN_LTE
*  A1 u A2 u A3 |- (M,Oi,Os) sat c1 lte c2
*
* FAILURE
* Fails unless the types are consistent among the three
* theorems.
***********************************************************)
fun EQN_LTE th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP eqn_lte th1) th2) th3;

(**********************************************************
* EQN_LT
*
```

```
* EQN_LT : thm -> thm -> thm -> thm
*
* SYNOPSIS
* Applies eqn_lt to theorems in the access-control logic
*
* DESCRIPTION
*
*        A1 |- (M,Oi,Os) sat c1 eqn n1
*        A2 |- (M,Oi,Os) sat c2 eqn n2
*        A3 |- (M,Oi,Os) sat n1 lt n2
*  -------------------------------------------- EQN_LT
*  A1 u A2 u A3 |- (M,Oi,Os) sat c1 lt c2
*
* FAILURE
* Fails unless the types are consistent among the three
* theorems.
***********************************************************)
fun EQN_LT th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP eqn_lt th1) th2) th3;

(***********************************************************
* EQN_EQN
*
* EQN_EQN : thm -> thm -> thm -> thm
*
* SYNOPSIS
* Applies eqn_eqn to theorems in the access-control logic
*
* DESCRIPTION
*
*        A1 |- (M,Oi,Os) sat c1 eqn n1
*        A2 |- (M,Oi,Os) sat c2 eqn n2
*        A3 |- (M,Oi,Os) sat n1 eqn n2
*  -------------------------------------------- EQN_EQN
*  A1 u A2 u A3 |- (M,Oi,Os) sat c1 eqn c2
*
* FAILURE
* Fails unless the types are consistent among the three
* theorems.
***********************************************************)
fun EQN_EQN th1 th2 th3 =
   MATCH_MP (MATCH_MP (MATCH_MP eqn_eqn th1) th2) th3;


(***********************************************************
* AND_SAYS_RL
*
* AND_SAYS_RL : thm -> thm
*
* SYNOPSIS
* Applies quoting rule to theorems in the access-control logic
*
* DESCRIPTION
*    th [(P says f) andf (Q says f)/A]
*  ------------------------- AND_SAYS_RL
*  th [P meet Q says f/A]
*
* FAILURE
* Fails unless the input theorem is of the form
* P says f  andf Q says f.
***********************************************************)
fun AND_SAYS_RL th = REWRITE_RULE [GSYM(SPEC_ALL And_Says_Eq)] th;

(***********************************************************
* AND_SAYS_LR
*
* AND_SAYS_LR : thm -> thm
```

```
*
* SYNOPSIS
* Applies And_Says rule to theorems in the access-control logic
*
* DESCRIPTION
*   th [P meet Q says f/A]
*  ------------------------------ AND_SAYS_LR
*   th [P says f andf Q says f/A]
*
* FAILURE
* Fails unless the input theorem is of the form P quoting Q
* says f.
**********************************************************)
fun AND_SAYS_LR th = REWRITE_RULE [SPEC_ALL And_Says_Eq] th;

(**********************************************************
* IDEMP_SPEAKS_FOR
*
* IDEMP_SPEAKS_FOR : term -> thm
*
* SYNOPSIS
* Specializes Idemp_Speaks_For to principal P
*
* DESCRIPTION
*
*  ------------------- IDEMP_SPEAKS_FOR P
*   |- P speaks_for P
*
* FAILURE
* Fails unless the term is a principal
**********************************************************)
fun IDEMP_SPEAKS_FOR term = ISPEC term (GEN ``P:'c Princ``(SPEC_ALL Idemp_Speaks_For));

(**********************************************************
* MONO_SPEAKS_FOR
*
* MONO_SPEAKS_FOR : thm -> thm -> thm
*
* SYNOPSIS
* Applies Mono_speaks_for to theorems in the access-control logic
*
* DESCRIPTION
*
*         A1 |- (M,Oi,Os) sat P speaks_for P'
*         A2 |- (M,Oi,Os) sat Q speaks_for Q'
*  ---------------------------------------------------------------- MONO_SPEAKS_FOR
*  A1 u A2 |- (M,Oi,Os) sat (P quoting Q) speaks_for (P' quoting Q')
*
* FAILURE
* Fails unless the types are consistent among the two
* theorems.
**********************************************************)
fun MONO_SPEAKS_FOR th1 th2 =
   (MATCH_MP (MATCH_MP Mono_speaks_for th1) th2);

(**********************************************************
* TRANS_SPEAKS_FOR
*
* TRANS_SPEAKS_FOR : thm -> thm -> thm
*
* SYNOPSIS
* Applies Trans_Speaks_For to theorems in the access-control logic
*
* DESCRIPTION
*
*    A1 |- (M,Oi,Os) sat P speaks_for Q
*    A2 |- (M,Oi,Os) sat Q speaks_for R
```

```
*  --------------------------------------- TRANS_SPEAKS_FOR
*  A1 u A2 |- (M,Oi,Os) sat P speaks_for R
*
* FAILURE
* Fails unless the types are consistent among the two
* theorems.
*********************************************************)
fun TRANS_SPEAKS_FOR th1 th2 =
   (MATCH_MP (MATCH_MP Trans_Speaks_For th1) th2);

(* ---------------------------------------------------------------------------- *)
(* EQF_ANDF1                                                                    *)
(*                                                                              *)
(* EQF_ANDF1 : thm -> thm -> thm                                                *)
(*                                                                              *)
(* SYNOPSIS                                                                     *)
(* Applies eqf_andf1 to substitute an equivalent term for another in the left *)
(* conjunct.                                                                    *)
(*                                                                              *)
(* DESCRIPTION                                                                  *)
(*                                                                              *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                              *)
(*    A2 |- (M,Oi,Os) sat f andf g                                              *)
(* ---------------------------------- EQF_ANDF1                                  *)
(* A1 u A2 |- (M,Oi,Os) sat f' andf g                                           *)
(*                                                                              *)
(* FAILURE                                                                      *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* a conjunction.  Fails unless all the types are consistent.                  *)
(* ---------------------------------------------------------------------------- *)
fun EQF_ANDF1 th1 th2 =
let
 val th3 = MATCH_MP eqf_andf1 th1
in
 MATCH_MP th3 th2
end

(* ---------------------------------------------------------------------------- *)
(* EQF_ANDF2                                                                    *)
(*                                                                              *)
(* EQF_ANDF2 : thm -> thm -> thm                                                *)
(*                                                                              *)
(* SYNOPSIS                                                                     *)
(* Applies eqf_andf2 to substitute an equivalent term for another in the left *)
(* conjunct.                                                                    *)
(*                                                                              *)
(* DESCRIPTION                                                                  *)
(*                                                                              *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                              *)
(*    A2 |- (M,Oi,Os) sat g andf f                                              *)
(* ---------------------------------- EQF_ANDF2                                  *)
(* A1 u A2 |- (M,Oi,Os) sat g andf f'                                           *)
(*                                                                              *)
(* FAILURE                                                                      *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* a conjunction.  Fails unless all the types are consistent.                  *)
(* ---------------------------------------------------------------------------- *)
fun EQF_ANDF2 th1 th2 =
let
 val th3 = MATCH_MP eqf_andf2 th1
in
 MATCH_MP th3 th2
end

(* ---------------------------------------------------------------------------- *)
(* EQF_CONTROLS                                                                 *)
(*                                                                              *)
```

```
(* EQF_CONTROLS : thm -> thm -> thm                                        *)
(*                                                                         *)
(* SYNOPSIS                                                                *)
(* Applies eqf_controls to substitute an equivalent formula f' for f in    *)
(* P controls f                                                            *)
(*                                                                         *)
(* DESCRIPTION                                                             *)
(*                                                                         *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                         *)
(*    A2 |- (M,Oi,Os) sat P controls f                                     *)
(* ----------------------------------- EQF_CONTROLS                        *)
(* A1 u A2 |- (M,Oi,Os) sat P controls f'                                  *)
(*                                                                         *)
(* FAILURE                                                                 *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* a conjunction.  Fails unless all the types are consistent.              *)
(* ----------------------------------------------------------------------- *)
fun EQF_CONTROLS th1 th2 =
let
 val th3 = MATCH_MP eqf_controls th1
in
 MATCH_MP th3 th2
end

(* ----------------------------------------------------------------------- *)
(* EQF_EQF1                                                                *)
(*                                                                         *)
(* EQF_EQF1 : thm -> thm -> thm                                            *)
(*                                                                         *)
(* SYNOPSIS                                                                *)
(* Applies eqf_eqf1 to substitute an equivalent term for another in the left  *)
(* side of the equivalence                                                 *)
(*                                                                         *)
(* DESCRIPTION                                                             *)
(*                                                                         *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                         *)
(*    A2 |- (M,Oi,Os) sat f eqf g                                          *)
(* ----------------------------------- EQF_EQF1                            *)
(* A1 u A2 |- (M,Oi,Os) sat f' eqf g                                       *)
(*                                                                         *)
(* FAILURE                                                                 *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* a equivalence.  Fails unless all the types are consistent.              *)
(* ----------------------------------------------------------------------- *)
fun EQF_EQF1 th1 th2 =
let
 val th3 = MATCH_MP eqf_eqf1 th1
in
 MATCH_MP th3 th2
end

(* ----------------------------------------------------------------------- *)
(* EQF_EQF2                                                                *)
(*                                                                         *)
(* EQF_EQF2 : thm -> thm -> thm                                            *)
(*                                                                         *)
(* SYNOPSIS                                                                *)
(* Applies eqf_eqf2 to substitute an equivalent term for another in the right *)
(* side of an equivalence.                                                 *)
(*                                                                         *)
(* DESCRIPTION                                                             *)
(*                                                                         *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                         *)
(*    A2 |- (M,Oi,Os) sat g eqf f                                          *)
(* ----------------------------------- EQF_EQF2                            *)
(* A1 u A2 |- (M,Oi,Os) sat g eqf f'                                       *)
(*                                                                         *)
```

```
(* FAILURE                                                                  *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* an equivalence.  Fails unless all the types are consistent.              *)
(* ------------------------------------------------------------------------ *)
fun EQF_EQF2 th1 th2 =
let
 val th3 = MATCH_MP eqf_eqf2 th1
in
 MATCH_MP th3 th2
end

(* ------------------------------------------------------------------------ *)
(* EQF_IMPF1                                                                *)
(*                                                                          *)
(* EQF_IMPF1 : thm -> thm -> thm                                            *)
(*                                                                          *)
(* SYNOPSIS                                                                 *)
(* Applies eqf_impf1 to substitute an equivalent term for another in the left *)
(* side of an implication                                                   *)
(*                                                                          *)
(* DESCRIPTION                                                              *)
(*                                                                          *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                          *)
(*    A2 |- (M,Oi,Os) sat f impf g                                          *)
(* ---------------------------------- EQF_IMPF1                             *)
(* A1 u A2 |- (M,Oi,Os) sat f' impf g                                       *)
(*                                                                          *)
(* FAILURE                                                                  *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* an implication.  Fails unless all the types are consistent.              *)
(* ------------------------------------------------------------------------ *)
fun EQF_IMPF1 th1 th2 =
let
 val th3 = MATCH_MP eqf_impf1 th1
in
 MATCH_MP th3 th2
end

(* ------------------------------------------------------------------------ *)
(* EQF_IMPF2                                                                *)
(*                                                                          *)
(* EQF_IMPF2 : thm -> thm -> thm                                            *)
(*                                                                          *)
(* SYNOPSIS                                                                 *)
(* Applies eqf_impf2 to substitute an equivalent term for another in the right*)
(* side of an implication.                                                  *)
(*                                                                          *)
(* DESCRIPTION                                                              *)
(*                                                                          *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                          *)
(*    A2 |- (M,Oi,Os) sat g impf f                                          *)
(* ---------------------------------- EQF_IMPF2                             *)
(* A1 u A2 |- (M,Oi,Os) sat g impf f'                                       *)
(*                                                                          *)
(* FAILURE                                                                  *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* an implication.  Fails unless all the types are consistent.              *)
(* ------------------------------------------------------------------------ *)
fun EQF_IMPF2 th1 th2 =
let
 val th3 = MATCH_MP eqf_impf2 th1
in
 MATCH_MP th3 th2
end

(* ------------------------------------------------------------------------ *)
(* EQF_NOTF                                                                 *)
```

```
(*                                                                          *)
(* EQF_NOTF : thm -> thm -> thm                                             *)
(*                                                                          *)
(* SYNOPSIS                                                                 *)
(* Applies eqf_notf to substitute an equivalent term for another in a       *)
(* negation.                                                                *)
(*                                                                          *)
(* DESCRIPTION                                                              *)
(*                                                                          *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                          *)
(*    A2 |- (M,Oi,Os) sat notf f                                            *)
(* ---------------------------------- EQF_NOTF                              *)
(* A1 u A2 |- (M,Oi,Os) sat notf f'                                         *)
(*                                                                          *)
(* FAILURE                                                                  *)
(* Fails unless the first theorem is an equivance and the second theorem is  *)
(* a negation.  Fails unless all the types are consistent.                  *)
(* ------------------------------------------------------------------------ *)
fun EQF_NOTF th1 th2 =
let
 val th3 = MATCH_MP eqf_notf th1
in
 MATCH_MP th3 th2
end

(* ------------------------------------------------------------------------ *)
(* EQF_ORF1                                                                 *)
(*                                                                          *)
(* EQF_ORF1 : thm -> thm -> thm                                             *)
(*                                                                          *)
(* SYNOPSIS                                                                 *)
(* Applies eqf_orf1 to substitute an equivalent term for another in the left *)
(* side of a disjunction.                                                   *)
(*                                                                          *)
(* DESCRIPTION                                                              *)
(*                                                                          *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                          *)
(*    A2 |- (M,Oi,Os) sat f orf g                                           *)
(* ---------------------------------- EQF_ORF1                              *)
(* A1 u A2 |- (M,Oi,Os) sat f' orf g                                        *)
(*                                                                          *)
(* FAILURE                                                                  *)
(* Fails unless the first theorem is an equivance and the second theorem is  *)
(* a disjunction. Fails unless all the types are consistent.                *)
(* ------------------------------------------------------------------------ *)
fun EQF_ORF1 th1 th2 =
let
 val th3 = MATCH_MP eqf_orf1 th1
in
 MATCH_MP th3 th2
end

(* ------------------------------------------------------------------------ *)
(* EQF_ORF2                                                                 *)
(*                                                                          *)
(* EQF_ORF2 : thm -> thm -> thm                                             *)
(*                                                                          *)
(* SYNOPSIS                                                                 *)
(* Applies eqf_orf2 to substitute an equivalent term for another in the right *)
(* side of a disjunction.                                                   *)
(*                                                                          *)
(* DESCRIPTION                                                              *)
(*                                                                          *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                          *)
(*    A2 |- (M,Oi,Os) sat g orf f                                           *)
(* ---------------------------------- EQF_ORF2                              *)
(* A1 u A2 |- (M,Oi,Os) sat g orf f'                                        *)
```

```
(*                                                                        *)
(* FAILURE                                                                *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* a disjunction. Fails unless all the types are consistent.              *)
(* ---------------------------------------------------------------------- *)
fun EQF_ORF2 th1 th2 =
let
 val th3 = MATCH_MP eqf_orf2 th1
in
 MATCH_MP th3 th2
end

(* ---------------------------------------------------------------------- *)
(* EQF_REPS                                                               *)
(*                                                                        *)
(* EQF_REPS : thm -> thm -> thm                                           *)
(*                                                                        *)
(* SYNOPSIS                                                               *)
(* Applies eqf_reps to substitute an equivalent formula for another in a  *)
(* a delegation formula.                                                  *)
(*                                                                        *)
(* DESCRIPTION                                                            *)
(*                                                                        *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                        *)
(*    A2 |- (M,Oi,Os) sat reps P Q f                                      *)
(* ---------------------------------- EQF_REPS                            *)
(* A1 u A2 |- (M,Oi,Os) sat reps P Q f'                                   *)
(*                                                                        *)
(* FAILURE                                                                *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* a delegation. Fails unless all the types are consistent.               *)
(* ---------------------------------------------------------------------- *)
fun EQF_REPS th1 th2 =
let
 val th3 = MATCH_MP eqf_reps th1
in
 MATCH_MP th3 th2
end

(* ---------------------------------------------------------------------- *)
(* EQF_SAYS                                                               *)
(*                                                                        *)
(* EQF_SAYS : thm -> thm -> thm                                           *)
(*                                                                        *)
(* SYNOPSIS                                                               *)
(* Applies eqf_says to substitute an equivalent formula for another in a  *)
(* a says formula.                                                        *)
(*                                                                        *)
(* DESCRIPTION                                                            *)
(*                                                                        *)
(*    A1 |- (M,Oi,Os) sat f eqf f'                                        *)
(*    A2 |- (M,Oi,Os) sat P says f                                        *)
(* ---------------------------------- EQF_SAYS                            *)
(* A1 u A2 |- (M,Oi,Os) sat P says f'                                     *)
(*                                                                        *)
(* FAILURE                                                                *)
(* Fails unless the first theorem is an equivance and the second theorem is   *)
(* a says formula. Fails unless all the types are consistent.             *)
(* ---------------------------------------------------------------------- *)
fun EQF_SAYS th1 th2 =
let
 val th3 = MATCH_MP eqf_says th1
in
 MATCH_MP th3 th2
end
```

```
(****************)
(**** Tactics ****)
(****************)

(***************************************************************************
ACL_CONJ_TAC

ACL_CONJ_TAC : (  a *term)   >(  a *term)list*(thm list  >thm))

SYNOPSIS
Reduces an ACL conjunctive goal to two separate subgoals.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat t1 andf t2, reduces it to the two sub-
goals corresponding to each conjunct separately.

        A ?- (M,Oi,Os) sat t1 andf t2
 ============================================= ACL_CONJ_TAC
           A ?- (M,Oi,Os) sat t1
           A ?- (M,Oi,Os) sat t2


FAILURE
Fails unless the conclusion of the goal is an ACL conjunction.
***************************************************************************)

fun ACL_CONJ_TAC (asl,term) =
let
  val (tuple,conj) = dest_sat term
  val (conj1,conj2) = dest_andf conj
  val conjTerm1 = mk_sat (tuple,conj1)
  val conjTerm2 = mk_sat (tuple,conj2)
in
 ([(asl,conjTerm1),(asl,conjTerm2)], fn [th1,th2] => ACL_CONJ th1 th2)
end


(***************************************************************************
ACL_DISJ1_TAC

ACL_DISJ1_TAC : (  a *term)   >((  a *term)list*(thm list  >thm))

SYNOPSIS
Selects the left disjunct of an ACL disjunctive goal.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat t1 orf t2, the tactic ACL_DISJ1_-
TAC reduces it to the subgoal corresponding to the left disjunct.

    A ?- (M,Oi,Os) sat t1 orf t2
  ================================ ACL_DISJ1_TAC
      A ?- (M,Oi,Os) sat t1


FAILURE
Fails unless the goal is an ACL disjunction.
***************************************************************************)

fun ACL_DISJ1_TAC (asl,term) =
let
  val (tuple,disj) = dest_sat term
  val (disj1,disj2) = dest_orf disj
  val disjTerm1 = mk_sat (tuple,disj1)
in
  ([(asl,disjTerm1)], fn [th] => ACL_DISJ1 disj2 th)
end
```

```
(***********************************************************************
ACL_DISJ2_TAC
ACL_DISJ2_TAC : (  a *term)   >((  a *term)list*(thm  l i s t  >thm))

SYNOPSIS
Selects the right disjunct of an ACL disjunctive goal.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat t1 orf t2, the tactic ACL_DISJ2_-
TAC reduces it to the subgoal corresponding to the right disjunct.

  A ?- (M,Oi,Os) sat t1 orf t2
 ================================ ACL_DISJ2_TAC
  A ?- (M,Oi,Os) sat t2

FAILURE
Fails unless the goal is an ACL disjunction.

***********************************************************************)
fun ACL_DISJ2_TAC (asl,term) =
let
  val (tuple,disj) = dest_sat term
  val (disj1,disj2) = dest_orf disj
  val disjTerm2 = mk_sat (tuple,disj2)
in
  ([(asl,disjTerm2)], fn [th] => ACL_DISJ2 disj1 th)
end

(***********************************************************************
ACL_MP_TAC

ACL_MP_TAC : t h m  >(  a *term)   >((  a *term)list*(thm  l i s t  >thm))

SYNOPSIS
Reduces a goal to an ACL implication from a known theorem.

DESCRIPTION
When applied to the theorem  A    |- (M,Oi,Os) sat s and the goal A ?- (M,Oi,Os)
sat t, the tactic ACL_MP_TAC reduces the goal to A ?- (M,Oi,Os) sat s impf t.
Unless  A    is a subset of A, this is an invalid tactic.

  A ?- (M,Oi,Os) sat t
 ================================ ACL_MP_TAC ( A    |- s)
  A ?- (M,Oi,Os) sat s impf t

FAILURE
Fails unless  A   is a subset of A.
***********************************************************************)

fun ACL_MP_TAC thb (asl,term) =
let
  val (tuple,form) = dest_sat term
  val (ntuple,nform) = dest_sat (concl thb)
  val newForm = mk_impf (nform,form)
  val newTerm = mk_sat (tuple,newForm)
  val predTerm = mk_sat (tuple,nform)
  val tupleType = type_of tuple
  val (_,[kripketype,_]) = dest_type tupleType
  val (_,[_,btype,_,_,_]) = dest_type kripketype
  val th2 = INST_TYPE [``:'b`` |-> btype] thb
in
    ([(asl,newTerm)], fn [th] => ACL_MP th2 th)
end

(***********************************************************************
ACL_AND_SAYS_RL_TAC
```

```
ACL_AND_SAYS_RL_TAC : (  a *term)   >((  a *term)list*(thm l i s t  >thm))

SYNOPSIS
Rewrites a goal with meet to two says statements.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat p meet q says f, returns a new sub-
goal in the form A ?- (M,Oi,Os) sat (p says f) andf (q says f).

  A ?- (M,Oi,Os) sat p meet q says f
 ===================================== ACL_AND_SAYS_RL_TAC
  A ?- (M,Oi,Os) sat (p says f)
                      andf (q says f)

FAILURE
Fails unless the goal is in the form p meet q says f.
***********************************************************************)

fun ACL_AND_SAYS_RL_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princs,prop) = dest_says form
 val (princ1,princ2) = dest_meet princs
 val conj1 = mk_says (princ1,prop)
 val conj2 = mk_says (princ2,prop)
 val conj = mk_andf (conj1,conj2)
 val newTerm = mk_sat (tuple,conj)
in
 ([(asl,newTerm)], fn [th] => AND_SAYS_RL th)
end

(***********************************************************************
ACL_AND_SAYS_LR_TAC

ACL_AND_SAYS_LR_TAC : (  a *term)   >((  a *term)list*(thm l i s t  >thm))

SYNOPSIS
Rewrites a goal with conjunctive says statements into a meet statement.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat (p says f) andf (q says f), re-
turns a new subgoal in the form A ?- (M,Oi,Os) sat p meet q says f.

  A ?- (M,Oi,Os) sat (p says f)
              andf (q says f)
 ======================================= ACL_AND_SAYS_LR_TAC
    A ?- (M,Oi,Os) sat p meet q says f

FAILURE
Fails unless the goal is in the form (p says f) andf (q says f).
************************************************************************)

fun ACL_AND_SAYS_LR_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (conj1,conj2) = dest_andf form
 val (princ1,prop) = dest_says conj1
 val (princ2,_) = dest_says conj2
 val princs = mk_meet (princ1,princ2)
 val newForm = mk_says (princs,prop)
 val newTerm = mk_sat (tuple,newForm)
in
 ([(asl,newTerm)], fn [th] => AND_SAYS_LR th)
end

(***********************************************************************
ACL_CONTROLS_TAC
```

```
ACL_CONTROLS_TAC : term >( a *term) >(( a *term)list*(thm list >thm))

SYNOPSIS
Reduces a goal to corresponding controls and says subgoals.

DESCRIPTION
When applied to a princ p and a goal A ?- (M,Oi,Os) sat f, returns a two new subgoals in the form A ?- (M,Oi
says f.

  A ?- (M,Oi,Os) sat f
 ====================================== ACL_CONTROLS_TAC p
  A ?- (M,Oi,Os) sat p controls f
  A ?- (M,Oi,Os) sat p says f

FAILURE
Fails unless the goal is a form type and p is a principle.
***********************************************************************)

fun ACL_CONTROLS_TAC princ (asl,term) =
let
 val (tuple,form) = dest_sat term
 val newControls = mk_controls (princ,form)
 val newTerm1 = mk_sat (tuple,newControls)
 val newSays = mk_says (princ,form)
 val newTerm2 = mk_sat (tuple,newSays)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2] => CONTROLS th1 th2)
end

(***********************************************************************
ACL_DC_TAC

ACL_DC_TAC : term >( a *term) >(( a *term)list*(thm list >thm))

SYNOPSIS
Reduces a goal to corresponding controls and speaks f or subgoals.

DESCRIPTION
When applied to a principal q and a goal A ?- (M,Oi,Os) sat p controls f, returns a two new subgoals in the

  A ?- (M,Oi,Os) sat p controls f
 ====================================== ACL_DC_TAC q
  A ?- (M,Oi,Os) sat p speaks_for q
  A ?- (M,Oi,Os) sat q controls f

FAILURE
Fails unless the goal is an ACL controls statement and q is a principle.
***********************************************************************)

fun ACL_DC_TAC princ2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ1,prop) = dest_controls form
 val formType = type_of form
 val speaksFor = ``(^princ1 speaks_for ^princ2):^(ty_antiq formType)``
 val newTerm1 = mk_sat (tuple,speaksFor)
 val newControls = mk_controls (princ2,prop)
 val newTerm2 = mk_sat (tuple,newControls)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2] => DC th1 th2)
end

(***********************************************************************
ACL_DOMI_TRANS_TAC

ACL_DOMI_TRANS_TAC : term >( a *term) >(( a *term)list*(thm list >thm))
```

```
SYNOPSIS
Reduces a goal to two subgoals using the transitive property of integrity levels.

DESCRIPTION
When applied to an integrity level l2 and a goal A ?- (M,Oi,Os) sat l1 domi l3, returns a two new subgoals in the f

  A ?- (M,Oi,Os) sat l1 domi l3
 ====================================== ACL_DOMI_TRANS_TAC l2
  A ?- (M,Oi,Os) sat l1 domi l2
  A ?- (M,Oi,Os) sat l2 domi l3

FAILURE
Fails unless the goal is an ACL domi statement and l2 is an integrity level.
******************************************************************************)

fun ACL_DOMI_TRANS_TAC iLev2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (iLev1,iLev3) = dest_domi form
 val formType = type_of form
 val newDomi1 = ``(^iLev1 domi ^iLev2):^(ty_antiq formType)``
 val newTerm1 = mk_sat (tuple,newDomi1)
 val newDomi2 = ``(^iLev2 domi ^iLev3):^(ty_antiq formType)``
 val newTerm2 = mk_sat (tuple,newDomi2)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2] => DOMI_TRANS th1 th2)
end

(******************************************************************************
ACL_DOMS_TRANS_TAC

ACL_DOMS_TRANS_TAC : t e r m >(  a *term)   >((  a *term)list*(thm l i s t  >thm))

SYNOPSIS
Reduces a goal to two subgoals using the transitive property of security levels.

DESCRIPTION
When applied to a security level l2 and a goal A ?- (M,Oi,Os) sat l1 doms l3,
returns a two new subgoals in the form A ?- (M,Oi,Os) sat l1 doms l2 and A ?-
(M,Oi,Os) sat l2 doms l3.

  A ?- (M,Oi,Os) sat l1 doms l3
 ====================================== ACL_DOMS_TRANS_TAC l2
  A ?- (M,Oi,Os) sat l1 doms l2
  A ?- (M,Oi,Os) sat l2 doms l3

FAILURE
Fails unless the goal is an ACL doms statement and l2 is a security level.
******************************************************************************)

fun ACL_DOMS_TRANS_TAC sLev2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (sLev1,sLev3) = dest_doms form
 val formType = type_of form
 val newDoms1 = ``(^sLev1 doms ^sLev2):^(ty_antiq formType)``
 val newTerm1 = mk_sat (tuple,newDoms1)
 val newDoms2 = ``(^sLev2 doms ^sLev3):^(ty_antiq formType)``
 val newTerm2 = mk_sat (tuple,newDoms2)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2] => DOMS_TRANS th1 th2)
end

(******************************************************************************
ACL_HS_TAC
```

```
ACL_HS_TAC : t e r m  >(   a *term)   >((   a *term)list*(thm  l i s t  >thm))

SYNOPSIS
Reduces a goal to two subgoals using the transitive property of ACL implications.

DESCRIPTION
When applied to an ACL formula f2 and a goal A ?- (M,Oi,Os) sat f1 impf f3,
returns a two new subgoals in the form A ?- (M,Oi,Os) sat f1 impf f2 and A ?-
(M,Oi,Os) sat f2 impf f3.

  A ?- (M,Oi,Os) sat f1 impf f3
 ======================================= ACL_HS_TAC f2
  A ?- (M,Oi,Os) sat f1 impf f2
  A ?- (M,Oi,Os) sat f2 impf f3

FAILURE
Fails unless the goal is an ACL implication and f2 is an ACL formula.
***********************************************************************)

fun ACL_HS_TAC f2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (f1,f3) = dest_impf form
 val newImpf1 = mk_impf (f1,f2)
 val newTerm1 = mk_sat (tuple,newImpf1)
 val newImpf2 = mk_impf (f2,f3)
 val newTerm2 = mk_sat (tuple,newImpf2)
in
 ([(asl,newTerm1),(asl,newTerm2)], fn [th1,th2] => HS th1 th2)
end

(***********************************************************************
ACL_IDEMP_SPEAKS_FOR_TAC

ACL_IDEMP_SPEAKS_FOR_TAC : (   a *term)   >((   a *term)list*(thm  l i s t  >thm))

SYNOPSIS
Proves a goal of the form p speaks for p.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat p speaks_for p, it will prove the goal.

  A ?- (M,Oi,Os) sat p speaks_for p
 ================================== ACL_IDEMP_SPEAKS_FOR_TAC

FAILURE
Fails unless the goal is an ACL formula of the form p speaks for p.
***********************************************************************)
fun ACL_IDEMP_SPEAKS_FOR_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ1,princ2) = dest_speaks_for form
 val th1 = IDEMP_SPEAKS_FOR princ1
 val tupleType = type_of tuple
 val (_,[kripketype,_]) = dest_type tupleType
 val (_,[_,btype,_,_,_]) = dest_type kripketype
 val formType = type_of form
 val (_,[proptype,princtype,inttype,sectype]) = dest_type formType
 val th2 = INST_TYPE [``:'a`` |-> proptype, ``:'b`` |-> btype, ``:'d`` |-> inttype, ``:'e`` |-> sectype] th1
in
 ([], fn xs => th2)
end

(***********************************************************************
ACL_IL_DOMI_TAC

ACL_IL_DOMI_TAC : t e r m  >t e r m  >(   a *term)   >((   a *term)list*(thm  l i s t  >thm))
```

```
SYNOPSIS
Reduces a goal comparing integrity levels of two principals to three subgoals.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat il q domi il p, integrity levels l2
and l1 it will return 3 subgoals.

  A ?- (M,Oi,Os) sat il q domi il p
 ===================================== ACL_IL_DOMI_TAC l2 l1
  A ?- (M,Oi,Os) sat l2 domi l1
  A ?- (M,Oi,Os) sat il q eqi l2
  A ?- (M,Oi,Os) sat il p eqi l1

FAILURE
Fails unless the goal is an ACL formula of the form il q domi il p.
***************************************************************************)
fun ACL_IL_DOMI_TAC ilev1 ilev2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val formtype = type_of form
 val (ilevprinc1,ilevprinc2) = dest_domi form
 val princ1eq = ``(^ilevprinc1 eqi ^ilev1):^(ty_antiq formtype)``
 val subgoal1 = mk_sat (tuple,princ1eq)
 val princ2eq = ``(^ilevprinc2 eqi ^ilev2):^(ty_antiq formtype)``
 val subgoal2 = mk_sat (tuple,princ2eq)
 val ilevdomi = ``(^ilev1 domi ^ilev2):^(ty_antiq formtype)``
 val subgoal3 = mk_sat (tuple,ilevdomi)
in
 ([(asl,subgoal1),(asl,subgoal2),(asl,subgoal3)], fn [th1,th2,th3] => IL_DOMI th2 th1 th3)
end

(***************************************************************************
ACL_MONO_SPEAKS_FOR_TAC

ACL_MONO_SPEAKS_FOR_TAC : (   a *term)   >((   a *term)list*(thm l i s t  >thm))

SYNOPSIS
Reduces a goal to corresponding speaks f or subgoals.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat (p quoting q) speaks_for ( p
quoting  q   ), it will return 2 subgoals.

  A ?- (M,Oi,Os) sat (p quoting q)
           speaks_for ( p    quoting  q   )
 =================================== ACL_MONO_SPEAKS_FOR_TAC
  A ?- (M,Oi,Os) sat p speaks_for  p
  A ?- (M,Oi,Os) sat q speaks_for  q

FAILURE
Fails unless the goal is an ACL formula of the form (p quoting q) speaks for ( p    quoting  q   ).
***************************************************************************)
fun ACL_MONO_SPEAKS_FOR_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val formtype = type_of form
 val (quote1,quote2) = dest_speaks_for form
 val (princ1,princ2) = dest_quoting quote1
 val (princ1',princ2') = dest_quoting quote2
 val speaksfor1 = ``(^princ1 speaks_for ^princ1'):^(ty_antiq formtype)``
 val subgoal1 = mk_sat (tuple,speaksfor1)
 val speaksfor2 = ``(^princ2 speaks_for ^princ2'):^(ty_antiq formtype)``
 val subgoal2 = mk_sat (tuple,speaksfor2)
in
 ([(asl,subgoal1),(asl,subgoal2)], fn [th1,th2] => MONO_SPEAKS_FOR th1 th2)
end
```

```
(************************************************************************
ACL_MP_SAYS_TAC

ACL_MP_SAYS_TAC : ( a *term) >(( a *term)list*(thm list >thm))

SYNOPSIS
Proves a goal of the form A ?- (M,Oi,Os) sat (p says (f1 impf f2)) impf
((p says f1) impf (p says f2))

DESCRIPTION
It will prove a goal of the following form: A ?- (M,Oi,Os) sat (p says (f1 impf f2)) impf ((p says f1) impf

  A ?- (M,Oi,Os) sat
      (p says (f1 impf f2)) impf
       ((p says f1) impf (p says f2))
 ======================================= ACL_MP_SAYS_TAC

FAILURE
Fails unless the goal is an ACL formula of the form (p says (f1 impf f2)) impf ((p says f1) impf (p says f2)
************************************************************************)
fun ACL_MP_SAYS_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (saysterm,_) = dest_impf form
 val (princ,impterm) = dest_says saysterm
 val (f1,f2) = dest_impf impterm
  val tupleType = type_of tuple
 val (_,[kripketype,_]) = dest_type tupleType
 val (_,[_,btype,_,_,_]) = dest_type kripketype
 val th1 = MP_SAYS princ f1 f2
 val th2 = INST_TYPE [``:'b`` |-> btype] th1
in
 ([], fn xs => th2)
end

(************************************************************************
ACL_QUOTING_LR_TAC

ACL_QUOTING_LR_TAC : ( a *term) >(( a *term)list*(thm list >thm))

SYNOPSIS
Reduces a says goal to corresponding quoting subgoal.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat p says q says f, it will return one
subgoal.

  A ?- (M,Oi,Os) sat p says q says f
 ======================================== ACL_QUOTING_LR_TAC
  A ?- (M,Oi,Os) sat p quoting q says f

FAILURE
Fails unless the goal is an ACL formula of the form p says q says f.
************************************************************************)
fun ACL_QUOTING_LR_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ1,saysterm) = dest_says form
 val (princ2,f) = dest_says saysterm
 val quotingterm = mk_quoting (princ1,princ2)
 val newform = mk_says (quotingterm,f)
 val subgoal = mk_sat (tuple,newform)
in
 ([(asl,subgoal)], fn [th] => QUOTING_LR th)
end
```

```
(***********************************************************************
ACL_QUOTING_RL_TAC

ACL_QUOTING_RL_TAC : (  a *term)   >((  a *term)list*(thm list  >thm))

SYNOPSIS
Reduces a quoting goal to corresponding says subgoal.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat p quoting q says f, it will return 1
subgoal.

  A ?- (M,Oi,Os) sat p quoting q says f
 ======================================== ACL_QUOTING_RL_TAC
  A ?- (M,Oi,Os) sat p says q says f

FAILURE
Fails unless the goal is an ACL formula of the form p quoting q says f.
***********************************************************************)
fun ACL_QUOTING_RL_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (quotingterm,f) = dest_says form
 val (princ1,princ2) = dest_quoting quotingterm
 val saysterm = mk_says (princ2,f)
 val newform = mk_says (princ1,saysterm)
 val subgoal = mk_sat (tuple,newform)
in
 ([(asl,subgoal)], fn [th] => QUOTING_RL th)
end

(***********************************************************************
ACL_REPS_TAC

ACL_REPS_TAC : term  >term  >(  a *term)   >((  a *term)list*(thm list  >thm))

SYNOPSIS
Reduces a goal to the corresponding reps subgoals.

DESCRIPTION
When applied to principals p, q and a goal A ?- (M,Oi,Os) sat f, it will return 3 subgoals.

  A ?- (M,Oi,Os) sat f
 ========================================= ACL_REPS_TAC p q
  A ?- (M,Oi,Os) sat q controls f
  A ?- (M,Oi,Os) sat p quoting q says f
  A ?- (M,Oi,Os) sat reps p q f

FAILURE
Fails unless the goal is an ACL formula.
***********************************************************************)
fun ACL_REPS_TAC princ1 princ2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val repterm = mk_reps (princ1,princ2,form)
 val subgoal1 = mk_sat (tuple,repterm)
 val quotingterm = mk_quoting (princ1,princ2)
 val saysterm = mk_says (quotingterm,form)
 val subgoal2 = mk_sat (tuple,saysterm)
 val controlsterm = mk_controls (princ2,form)
 val subgoal3 = mk_sat (tuple,controlsterm)
in
 ([(asl,subgoal1),(asl,subgoal2),(asl,subgoal3)], fn [th1,th2,th3] => REPS th1 th2 th3)
end

(***********************************************************************
ACL_REP_SAYS_TAC
```

```
ACL_REP_SAYS_TAC : term >( a *term) >(( a *term)list*(thm list >thm))

SYNOPSIS
Reduces a says goal to the corresponding reps subgoals.

DESCRIPTION
When applied to principal p and a goal A ?- (M,Oi,Os) sat q says f, it will return two subgoals.

  A ?- (M,Oi,Os) sat q says f
 ======================================= ACL_REP_SAYS_TAC p
  A ?- (M,Oi,Os) sat p quoting q says f
  A ?- (M,Oi,Os) sat reps p q f

FAILURE
Fails unless the goal is an ACL formula in the form of q says f.
***********************************************************************)
fun ACL_REP_SAYS_TAC princ1 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ2,f) = dest_says form
 val repsterm = mk_reps (princ1,princ2,f)
 val subgoal1 = mk_sat (tuple,repsterm)
 val quotingterm = mk_quoting (princ1,princ2)
 val saysterm = mk_says (quotingterm,f)
 val subgoal2 = mk_sat (tuple,saysterm)
in
 ([(asl,subgoal1),(asl,subgoal2)], fn [th1,th2] => REP_SAYS th1 th2)
end

(***********************************************************************
ACL_SAYS_TAC

ACL_SAYS_TAC : ( a *term) >(( a *term)list*(thm list >thm))

SYNOPSIS
Reduces a says goal to the corresponding subgoal.

DESCRIPTION
When applied to a goal A ?- (M,Oi,Os) sat p says f, it will return one subgoal.

  A ?- (M,Oi,Os) sat p says f
 ============================== ACL_SAYS_TAC
  A ?- (M,Oi,Os) sat f

FAILURE
Fails unless the goal is an ACL formula in the form of p says f.
***********************************************************************)
fun ACL_SAYS_TAC (asl,term) =
let
 val (tuple,form) = dest_sat term
 val (princ,f) = dest_says form
 val subgoal = mk_sat (tuple,f)
in
 ([(asl,subgoal)], fn [th] => SAYS princ th)
end

(***********************************************************************
ACL_SPEAKS_FOR_TAC

ACL_SPEAKS_FOR_TAC : term >( a *term) >(( a *term)list*(thm list >thm))

SYNOPSIS
Reduces a says goal to the corresponding says and speaks_for subgoals.

DESCRIPTION
When applied to a principal p and a goal A ?- (M,Oi,Os) sat q says f, it will return two subgoals.
```

```
  A ?- (M,Oi,Os) sat q says f
 =================================== ACL_SPEAKS_FOR_TAC p
  A ?- (M,Oi,Os) sat p says f
  A ?- (M,Oi,Os) sat p speaks_for q

FAILURE
Fails unless the goal is an ACL formula in the form of p says f.
****************************************************************************)
fun ACL_SPEAKS_FOR_TAC princ2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val formtype = type_of form
 val (princ1,f) = dest_says form
 val newSpeaksfor = ``(^princ2 speaks_for ^princ1):^(ty_antiq formtype)``
 val newTerm1 = mk_sat (tuple,newSpeaksfor)
  val newSays = mk_says (princ2,f)
 val newTerm2 = mk_sat (tuple,newSays)
in
 ([(asl,newTerm1),(asl,newTerm2)],fn [th1,th2] => SPEAKS_FOR th1 th2)
end

(****************************************************************************
ACL_TRANS_SPEAKS_FOR_TAC

ACL_TRANS_SPEAKS_FOR_TAC : t e r m >(  a *term)   >((  a *term)list*(thm l i s t  >thm))

SYNOPSIS
Reduces a speaks_for goal to two corresponding speaks_for subgoals, using the transitive property of speaks_for.

DESCRIPTION
When applied to a principal q and a goal A ?- (M,Oi,Os) sat p speaks_for r, it
will return two subgoals.

  A ?- (M,Oi,Os) sat p speaks_for r
 =================================== ACL_TRANS_SPEAKS_FOR_TAC q
  A ?- (M,Oi,Os) sat q speaks_for r
  A ?- (M,Oi,Os) sat p speaks_for q

FAILURE
Fails unless the goal is an ACL formula in the form of p speaks for r.
****************************************************************************)
fun ACL_TRANS_SPEAKS_FOR_TAC princ2 (asl,term) =
let
 val (tuple,form) = dest_sat term
 val formtype = type_of form
 val (princ1,princ3) = dest_speaks_for form
 val newSpeaksFor1 = ``(^princ1 speaks_for ^princ2):^(ty_antiq formtype)``
 val newTerm1 = mk_sat (tuple,newSpeaksFor1)
 val newSpeaksFor2 = ``(^princ2 speaks_for ^princ3):^(ty_antiq formtype)``
 val newTerm2 = mk_sat (tuple,newSpeaksFor2)
in
 ([(asl,newTerm1),(asl,newTerm2)],fn [th1,th2] => TRANS_SPEAKS_FOR th1 th2)
end


end; (* structure *)
```

D.6 acl_infRules.sig

```
(* File: acl_infRules.sig created 2/19/2009 *)
(* Author: Shiu-Kai Chin, skchin@syr.edu *)

signature acl_infRules =
sig

type tactic = Abbrev.tactic;
type thm_tactic = Abbrev.thm_tactic;
type conv = Abbrev.conv;
type thm = Thm.thm;
type term = Term.term;

val ACL_TAUT_TAC : tactic;
val ACL_TAUT : term -> thm;
val ACL_ASSUM : term -> thm;
val ACL_ASSUM2 : term -> term -> term -> thm;
val ACL_MP : thm -> thm -> thm;
val ACL_MT : thm -> thm -> thm
val ACL_SIMP1 : thm -> thm;
val ACL_SIMP2 : thm -> thm;
val ACL_CONJ : thm -> thm -> thm;
val ACL_DISJ1 : term -> thm -> thm;
val ACL_DISJ2 : term -> thm -> thm;
val CONTROLS : thm -> thm -> thm;
val REPS : thm -> thm -> thm -> thm;
val REP_SAYS : thm -> thm -> thm;
val ACL_DN : thm -> thm;
val SAYS : term -> thm -> thm;
val MP_SAYS : term -> term -> term -> thm;
val SPEAKS_FOR : thm -> thm -> thm;
val HS : thm -> thm -> thm;
val DC : thm -> thm -> thm;
val SAYS_SIMP1 : thm -> thm;
val SAYS_SIMP2 : thm -> thm;
```

```
val DOMI_TRANS : thm -> thm -> thm;

val DOMS_TRANS : thm -> thm -> thm;

val IL_DOMI : thm -> thm -> thm -> thm;

val SL_DOMS : thm -> thm -> thm -> thm;

val QUOTING_RL : thm -> thm;

val QUOTING_LR : thm -> thm;

val EQN_LTE : thm -> thm -> thm -> thm;

val EQN_LT : thm -> thm -> thm -> thm;

val EQN_EQN : thm -> thm -> thm -> thm;

val AND_SAYS_RL : thm -> thm;

val AND_SAYS_LR : thm -> thm;

val IDEMP_SPEAKS_FOR : term -> thm;

val MONO_SPEAKS_FOR : thm -> thm -> thm;

val TRANS_SPEAKS_FOR : thm -> thm -> thm;

val EQF_ANDF1 : thm -> thm -> thm

val EQF_ANDF2 : thm -> thm -> thm

val EQF_CONTROLS : thm -> thm -> thm

val EQF_EQF1 : thm -> thm -> thm

val EQF_EQF2 : thm -> thm -> thm

val EQF_IMPF1 : thm -> thm -> thm

val EQF_IMPF2 : thm -> thm -> thm

val EQF_NOTF : thm -> thm -> thm

val EQF_ORF1 : thm -> thm -> thm

val EQF_ORF2 : thm -> thm -> thm

val EQF_REPS : thm -> thm -> thm

val EQF_SAYS : thm -> thm -> thm

val ACL_CONJ_TAC : 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_DISJ1_TAC : 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_DISJ2_TAC : 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_MP_TAC : thm -> 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_AND_SAYS_RL_TAC : 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_AND_SAYS_LR_TAC : 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_CONTROLS_TAC : term -> 'a * term -> ('a * term) list * (thm list -> thm)
```

```
val ACL_DC_TAC : term -> 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_DOMI_TRANS_TAC : term -> 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_DOMS_TRANS_TAC : term -> 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_HS_TAC : term -> 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_IDEMP_SPEAKS_FOR_TAC : 'a * term -> 'b list * ('c -> thm)

val ACL_IL_DOMI_TAC : term -> term -> 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_MONO_SPEAKS_FOR_TAC : 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_MP_SAYS_TAC : 'a * term -> 'b list * ('c -> thm)

val ACL_QUOTING_LR_TAC : 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_QUOTING_RL_TAC : 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_REPS_TAC : term -> term -> 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_REP_SAYS_TAC : term -> 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_SAYS_TAC : 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_SPEAKS_FOR_TAC : term -> 'a * term -> ('a * term) list * (thm list -> thm)

val ACL_TRANS_SPEAKS_FOR_TAC : term -> 'a * term -> ('a * term) list * (thm list -> thm)

end;
```

D.7 ante_allTacs.sml

```
(* File: ante_allTacs.sml, utility tactics, rules, conversionals. *)
(* Author: F. Lockwood Morris  <lockwood@ecs.syr.edu> *)
(* 5/4/03: merge in all of ante_appTacs, and und_asm *)
(* 8/13/02: adapt to HOL-4 *)
(* 7/27/02: made a structure, with a signature *)
(* 4/12/01: XL_FUN_EQ_CONV *) (* 3/23/01: added lines, asm, with_asm *)
(* 1/29/01: rearrangement in 2 (more and less elementary) parts *)
(* 1/26/01: reconciling with ante_allDesc.tex, for semi-public release *)
(* 11/16.00: added symOfEqn, STRIP_EXISTS_UNIQUE_TAC *)
(* 6/6/00; CONJ_DISCH_TAC will now fail for non-implications. *)
(* 3/15/00: Starting 6-comp. cat. rework. Intro. shorter name GCONJ_LIST*)
(* 3/11/00: added IMP_RES_THENL, IMP_RES_ASM_THENL, GCONJ_TRIP *)
(* 3/1/00: added short-name tactics AR, CRE, CR, CREL, CRL *)
(* 2/16/00: added ASM_MATCH_THEN (superceded 2/17 by PAT_ASSUM), TR_TAC *)
(* 1/22/00: LEMMA_TAC, LEMMA_MP_TAC now based on SUBGOAL_THEN *)
(* 1/8/00: DUP_ARGS_TAC added *) (* 1/5/00: EXISTS_UNIQUE_TAC added *)

structure ante_allTacs :> ante_allTacs =
struct

open HolKernel boolLib;
(* app load ["res_quanLib","Cond_rewrite","pairLib","pred_setTheory"];*)
open res_quanLib res_quanTheory Cond_rewrite pairTheory pairLib
     pred_setTheory Parse;
infix 2 THENSGS THENFIN;
```

```
(* ****** FIRST PART: very elementary rules, tactics, conversions ****** *)

(* Embryonic model for isThing_TACs: CUMUL_CONJ_TAC, capturing the idea:
   "To prove C1 /\ C2, it is enough to prove C1 and to prove C1 ==> C2." *)

val cumul_thm = prove (Term`A /\ (A ==> B) ==> A /\ B`,
REPEAT STRIP_TAC THEN RES_TAC THEN ASM_REWRITE_TAC []);

val CUMUL_CONJ_TAC = MATCH_MP_TAC cumul_thm THEN CONJ_TAC;

(* Tactic: T_TAC, solves only ?-T, use after, e.g., COND_RW_TAC. It is
  hoped to be, if not quite documentation, at least more informative to
  use any of T_TAC, REFL_TAC, and TR_TAC when it is enough to finish
  solving a goal, even though REWRITE_TAC [] will outdo them all; also
  they should be cheaper than REWRITE_TAC [] to use with TRY in building
  other tactics, such as CRE, CR, CREL, CRL below. *)

val T_TAC = ACCEPT_TAC TRUTH;

(* TR_TAC, combines T_TAC, REFL_TAC, and REPEAT CONJ_TAC to sweep up
  after COND_REWRITE1_TAC and COND_RW_TAC have done all the work. *)

val TR_TAC = REPEAT CONJ_TAC THEN (T_TAC ORELSE REFL_TAC);

(* Tactics to drop an antecedent or an asm after it has served its turn.*)

val ANTE_DROP_TAC = DISCH_THEN (fn _ => ALL_TAC);

fun UNASSUME_TAC th = UNDISCH_TAC (concl th) THEN ANTE_DROP_TAC;

fun ulist x = [x]; (* most useful for thm-tactic REWRITE_TAC o ulist *)

(* (thm-tactics -> tactic)s built on ANTE_CONJ_CONV and extracting one half
   the antecedent; used the by tactics below, which may serve as
   documentation, but also, like DISCH_THEN, useful raw. *)

fun HALF_DISCH_THEN ttac = CONV_TAC ANTE_CONJ_CONV THEN DISCH_THEN ttac;

fun SWAP_HALF_DISCH_THEN ttac = DISCH_THEN
    (MP_TAC o CONV_RULE (REWR_CONV CONJ_SYM)) THEN HALF_DISCH_THEN ttac;

(* A tactic to make the characteristic use of ANTE_CONJ_CONV, and another
   just like it, but swaps the two antecedents first, that is:

A ?- u /\ v ==> w                    A ?- u /\ v ==> w
================== HALF_DISCH_TAC    ================== SWAP_HALF_DISCH_TAC
A u {u} ?- v ==> w                   A u {v} ?- u ==> w                  *)

val HALF_DISCH_TAC = HALF_DISCH_THEN ASSUME_TAC;

val SWAP_HALF_DISCH_TAC = SWAP_HALF_DISCH_THEN ASSUME_TAC;

(* Some conversional analogues of Lisp’s CAAR, CDAR, CADR, CDDR. *)

val LLAND_CONV = LAND_CONV o LAND_CONV;
val LRAND_CONV = LAND_CONV o RAND_CONV;
val RLAND_CONV = RAND_CONV o LAND_CONV;
val RRAND_CONV = RAND_CONV o RAND_CONV;

(* Tactic to make up for bad planning: reverses an equational assumption *)

fun ASM_SYM_TAC t = UNDISCH_TAC t
                    THEN DISCH_THEN (ASSUME_TAC o GSYM);

(* A pure abbreviation of the rewrite I do most often. *)

val AR = ASM_REWRITE_TAC [];
```

```
(* A pair of tactics, LEMMA_TAC, LEMMA_MP_TAC: term -> tactic, for
   explicitly introducing a lemma as a separate subgoal to be proved;
   the original goal can then be tackled with the lemma as an added
   assumption (LEMMA_TAC) or hypothesis (LEMMA_MP_TAC).  These are all
   easy instances of SUBGOAL_THEN, but it took me a long time to
   understand its documentation, at least the first sentence of which
   applies as well to these tactics: "The user proposes a lemma and is
   then invited to prove it under the current assumptions."

         A ?- G                              A ?- G
  ===================== LEMMA_TAC L   ==================== LEMMA_MP_TAC L
  A ?- L   A u {L} ?- G               A ?- L  A ?- L ==> G             *)

fun LEMMA_MP_TAC L = SUBGOAL_THEN L MP_TAC;
fun LEMMA_TAC L = SUBGOAL_THEN L STRIP_ASSUME_TAC;

(* An abbreviatory rule for drawing an inference from 2 thms by a third. *)

fun MATCH_MP2 T_IMP_IMP T1 T2 = MATCH_MP (MATCH_MP T_IMP_IMP T1) T2;

(* Sometimes one breaks up an equivalence goal with EQ_TAC merely in
 order to infer some easy conclusion from each side and add it to the
 assumptions; one would then wish that the two implicational subgoals
 could be recombined. The following tactic simulates that effect:

                         A ?- B = C
        ================================================= EQ_HYP_TAC H
        A ?- (B ==> H) /\ (C ==> H)      A u {H} ?- B = C              *)

val EQ_HYP_TAC_LEM = prove (Term
`!H B C. ((B ==> H) /\ (C ==> H)) /\ (H ==> (B = C)) ==> (B = C)`,
REPEAT STRIP_TAC THEN EQ_TAC THEN DISCH_TAC
THEN RES_TAC THEN RES_TAC);

fun EQ_HYP_TAC h = MATCH_MP_TAC (SPEC h EQ_HYP_TAC_LEM)
 THEN (CONJ_TAC THENL [ALL_TAC, STRIP_TAC]);

(* Gordon's line-assumption selection technique, Intro. to HOL p. 55 *)

fun lines tok = let val wt = words2 "␣" tok in (fn t =>
 let val x = #Name (Rsyntax.dest_var
                    (rator (lhs (#Body (Rsyntax.dest_forall t)))))
 in mem x wt end handle _ => false) end;

(* Similar idea to pick up one assumption, of arbitrary structure, by
 a string containing enough of its variable occurrences, free and bound
 indifferently, in order from the left to identify it uniquely.
 5/4/03: jiggered to observe left-to-right rule even for terms with :: . *)

fun asm vars t =
 let exception MAT and NOMAT;
  fun mat [] t = raise MAT
    | mat (vvl as (v :: vl)) t =
       if is_var t then if #Name (Rsyntax.dest_var t) = v then vl
                                                             else raise NOMAT
       else if is_const t then vvl
       else if is_comb t then
         if is_abs (rand t) andalso
             (is_res_abstract t orelse is_res_forall t
              orelse is_res_exists t orelse is_res_select t
              orelse is_res_exists_unique t)
           then let val (var, P, t') =
                        if is_res_abstract t then dest_res_abstract t
                        else if is_res_forall t then dest_res_forall t
                        else if is_res_exists t then dest_res_exists t
                        else if is_res_select t then dest_res_select t
```

```
                    else dest_res_exists_unique t
                in mat (mat (mat vvl var) P) t' end
         else mat (mat vvl (rator t)) (rand t)
       else mat (mat vvl (#Bvar (Rsyntax.dest_abs t)))
                (#Body (Rsyntax.dest_abs t));
 in mat (words2 "␣" vars) t = [] handle MAT => true | NOMAT => false end;

(* Function with_asm: string -> thm-tactic -> tactic applies its thm-tactic
  argument to the first assumption accepted by (asm string). *)

fun with_asm s ttac = ASSUM_LIST (ttac o first (asm s o concl));

fun und_asm s = with_asm s (UNDISCH_TAC o concl);

fun drop_asm s = with_asm s UNASSUME_TAC;

(* tty = Toggle show_TYpes [+ meaningless nostalgia for the teletype] *)

fun tty () = (show_types := not (!show_types); !show_types);

(* utility functions  for terms - recognize given unary/binary operator *)

fun is_unap string t =  is_comb t andalso
                        is_const (rator t) andalso
                        #Name (Rsyntax.dest_const (rator t)) = string;

fun is_binap string t = is_comb t andalso is_unap string (rator t);

(* REV_EXISTS_CONV generalizes SWAP_EXISTS_CONV, reverses the order of any
 number of existential quantifiers, and REV_FORALL_CONV does the same for
 universals. Both do nothing if there is one quantifier, fail if none. *)

fun REV_EXISTS_CONV t =
 let fun bury_outer t = (if is_exists (#Body (Rsyntax.dest_exists t))
                        then SWAP_EXISTS_CONV
                             THENC RAND_CONV (ABS_CONV bury_outer)
                        else ALL_CONV) t
 in (if is_exists (#Body (Rsyntax.dest_exists t))
    then RAND_CONV (ABS_CONV REV_EXISTS_CONV)
         THENC bury_outer
    else ALL_CONV) t end;

val REV_FORALL_CONV =
 let fun not_forall_iter t = (* t is a negation *)
    (if is_forall (rand t) then NOT_FORALL_CONV
                               THENC RAND_CONV (ABS_CONV not_forall_iter)
                          else ALL_CONV) t
     and exists_not_iter t = (if is_exists t
                             then RAND_CONV (ABS_CONV exists_not_iter)
                                  THENC EXISTS_NOT_CONV
                             else ALL_CONV) t
 in REWR_CONV (GSYM (CONJUNCT1 NOT_CLAUSES))
    THENC RAND_CONV (not_forall_iter THENC REV_EXISTS_CONV
                    THENC exists_not_iter)
    THENC REWR_CONV (CONJUNCT1 NOT_CLAUSES) end;

(* ***** SECOND PART: less elementary rules, tactics, conversions ****** *)

(* A tactical for replicating a tactic, for use with THENL, and a variant
   which gives one of a pair of arguments to each side. *)

fun DUP_TAC tac = [tac, tac];

fun DUP_ARGS_TAC (p,q) a_tac = [a_tac p, a_tac q];

(* A slight generalization of Dan Zhou's rule, 'IMP_CONJ_LIST', intended
   like it to assist in putting theorems into a form acceptable to
```

```
   COND_REWRITE1_TAC and _CONV. The following rule takes as 1st argument
   a rule for producing a list of theorems from a theorem, and produces
   a rule that applies the argument rule at the bottom of any nest of
   universal quantifiers and implications, restoring the quantifiers and
   hypotheses to each element of the resulting list of theorems. *)

fun FORALL_IMP_LIST_RULE lisrul th =
  if is_forall (concl th)
   then let val (v, th') = SPEC_VAR th
        in map (GEN v) (FORALL_IMP_LIST_RULE lisrul th') end
  else if is_imp (concl th)
   then let val {ant=a, conseq=c} = Rsyntax.dest_imp (concl th)
        in map (DISCH a) (FORALL_IMP_LIST_RULE lisrul (UNDISCH th)) end
   else lisrul th;

(* The next rule reproduces what Dan Zhou's IMP_CONJ_LIST does, but it
   removes and replaces universal quantifiers as well.

   GCONJ_LIST: int -> thm -> thm list

        A |- a ==> b ==> !x ... ==> (e1 /\ e2 /\ ... /\ en)
  ------------------------------------------------------ GCONJ_LIST n
            [A |- a ==> b ==> !x ... ==> e1,
             A |- a ==> b ==> !x ... ==> e2, ... ,
             A |- a ==> b ==> !x ... ==> en]

FAILURE: Fails if the integer argument (n) is less than one, or if the
         final conclusion of input theorem has less than n conjuncts.   *)

fun GCONJ_LIST n = FORALL_IMP_LIST_RULE (CONJ_LIST n);

(* Rules to turn a Boolean eqn. into lhs ==> rhs, resp. rhs ==> lhs *)

val impOfEqn = hd o FORALL_IMP_LIST_RULE (ulist o fst o EQ_IMP_RULE);

val impByOfEqn = hd o FORALL_IMP_LIST_RULE (ulist o snd o EQ_IMP_RULE);

(* The built-in GSYM occasionally does too much, reversing equations in
 the hypotheses which should be left alone; hence the following: *)

val symOfEqn = hd o FORALL_IMP_LIST_RULE (ulist o SYM);

(* Following conversion, GCONJ_CONV, offers the virtues
   of GCONJ_LIST in a conversion, hence potentially in a tactic
   as well as a rule. AND_IMP_THM or its conversion may exist under
   another name, but I can't find it. Object of GCONJ_CONV
   is to drag conjunctions to the top level, duplicating the antecedents
   and universal quantifiers which had been above them. The two
   subsidiary conversion have an obvious common pattern, and so could
   be written as applications of a conversional, say BINOP_ITER_CONV,
   but the latter seems as if it would be hard to explain. No parameter
   n here; GCONJ_CONV just breaks up all the top-level /\'s it finds,
   by recursion on the right operand, at the bottom of a ! - ==> nest. *)

val AND_IMP_THM = prove (Term`(H ==> B /\ C) = (H ==> B) /\ (H ==> C)`,
EQ_TAC THEN REPEAT STRIP_TAC THEN RES_TAC);

fun IMP_CONJ_CONV t =
 ((REWR_CONV AND_IMP_THM THENC RAND_CONV IMP_CONJ_CONV)
  ORELSEC ALL_CONV) t;

fun FORALL_CONJ_CONV t =
 ((FORALL_AND_CONV THENC RAND_CONV FORALL_CONJ_CONV)
  ORELSEC ALL_CONV) t;

fun GCONJ_CONV t =
 (if is_imp t then RAND_CONV GCONJ_CONV
```

```
                    THENC IMP_CONJ_CONV
  else if is_forall t then RAND_CONV (ABS_CONV GCONJ_CONV)
                            THENC FORALL_CONJ_CONV
  else ALL_CONV) t;

(* Following rule, GCONJUNCTS, can sometimes replace GCONJ_LIST n, when n
 is the number of conjuncts that GCONJ_CONV will find (unlike CONJUNCTS,
 it does not recurse to the left). Only reason to bother with this is that
 it avoids SPEC_VAR, used by GCONJ_LIST, and temporarilly buggy with the
 advent of HOL-4, August '02. *)

val GCONJUNCTS = CONJUNCTS o CONV_RULE GCONJ_CONV;

(* Special rules for 2 and for 3 conjuncts *)

val GCONJ_PAIR = CONJ_PAIR o CONV_RULE GCONJ_CONV;

fun GCONJ_TRIP th = let val cj = CONV_RULE GCONJ_CONV th;
                        val (a, b) = CONJ_PAIR cj; val (c, d) = CONJ_PAIR b
                    in (a, c, d) end;

(* Next rule makes non-equational impl'ns usable by COND_REWRITE1_TAC *)

(* eqeqt = |- t = (t = T) ------- CAUTION, use only with ONCE_rules etc. *)

val eqeqt = SYM (hd (tl (CONJ_LIST 4 (SPEC_ALL EQ_CLAUSES))));

val FORALL_IMP_EQT_RULE =
 hd o FORALL_IMP_LIST_RULE (ulist o PURE_ONCE_REWRITE_RULE [eqeqt]);

(* Front end, COND_RW_TAC, for COND_REWRITE1_TAC to make it aim at
   rewriting a non-equation (usually) to T. *)

fun COND_RW_TAC th = COND_REWRITE1_TAC (FORALL_IMP_EQT_RULE th);

(* Tacticals THENSGS, THENFIN to be used with a tactic like
   COND_REWRITE1_TAC which spawns an indeterminate number of subsidiary
   goals and finally a transformed main goal; the function then_sgs_fin
   is a bifurcating THEN applying different tactics to the two kinds of
   resulting goal, and the infix THENFIN and THENSGS specialize this. *)

(* If just two subgoals are generated, prefer THEN1 to THENSGS. *)

fun bisect 0 xs = ([], xs)  (* unappend, with given length of 1st piece *)
  | bisect n xxs = let val (l1, l2) = bisect (n-1) (tl xxs)
                       in (hd xxs :: l1, l2) end;

(* mapshape, used in Gordon and Melham to program THEN, is not provided
   under that name in this system. We imitate G&M pp. 381, 383. *)

fun mapshape [] _ _ = []
  | mapshape mms ffs xs =
     let val (m, ms, f, fs) = (hd mms, tl mms, hd ffs, tl ffs);
         val (ys, zs) = bisect m xs
     in f ys :: mapshape ms fs zs end;

fun then_sgs_fin (T1, Tsgs, Tmg) g =
 let val (gl, p) = T1 g;
 in if null gl then ([], p) else
     let val (sgs, mgl) = bisect (length gl - 1) gl;
         val (gll, pl) = split (map Tsgs sgs);
         val (glm, pm) = Tmg (hd mgl);
         val (gll', pl') = (gll @ [glm], pl @ [pm])
     in (flatten gll', (p o mapshape (map length gll') pl'))
 end end;

fun T1 THENSGS T2 = then_sgs_fin (T1, T2, ALL_TAC);
```

```
fun T1 THENFIN T2 = then_sgs_fin (T1, ALL_TAC, T2);

(* Following four definitions are largely to give short names to workhorse
  tactics, but we avoid precise synonyms. CRE, CR are like
  COND_REWRITE1_TAC, COND_RW_TAC respectively (in particular they fail if
  they find no match) but they may solve the final (sub)goal completely if
  it is reduced respectively to a reflexive equation or to T. (Mnemonics:
  CR: "conditional rewriting",  E: "with an equation".)
     CREL, CRL take a list of conditional equations (resp. non-equations)
  and try rewriting once with each, from left to right in the list, working
  always on only the final subgoal produced by what has gone before, and
  at the end optimistically try to polish it off with TR_TAC. Note that
  there is no direct way to mix equational and non-equational theorems in
  one list, but alternating calls of CREL and CRL, joined by THENFIN,
  will do the job.                                                     *)

fun CRE th = COND_REWRITE1_TAC th THENFIN TRY REFL_TAC;
fun CR th = COND_RW_TAC th THENFIN TRY T_TAC;

fun CREL [] = TRY TR_TAC
  | CREL (th :: ths) = TRY (COND_REWRITE1_TAC th) THENFIN CREL ths;

fun CRL [] = TRY TR_TAC
  | CRL (th :: ths) = TRY (COND_RW_TAC th) THENFIN CRL ths;

(* A rule: GEXT, like EXT, but strips all the variables from
   !x...z. F x ... z = G x ... z. *)

fun GEXT th =
 let fun iter_ext nil thm = thm
       | iter_ext (v :: l) thm = EXT (GEN v (iter_ext l thm));
     val (vars, _) = strip_forall (concl th)
 in iter_ext vars (SPEC_ALL th) end;

(* Iterated version, XL_FUN_EQ_CONV, of X_FUN_EQ_CONV for a sequence of
  specified variables, and corresp. tactic XL_FUN_EQ_TAC, which strips
  off the resulting universal quantifiers. *)

fun XL_FUN_EQ_CONV [] = ALL_CONV
  | XL_FUN_EQ_CONV (v :: vs) = X_FUN_EQ_CONV v THENC
                               QUANT_CONV (XL_FUN_EQ_CONV vs);

fun XL_FUN_EQ_TAC vl = CONV_TAC (XL_FUN_EQ_CONV vl) THEN
                       MAP_EVERY (fn _ => GEN_TAC) vl;

(* A theorem to support XP_FUN_EQ_CONV *)

val pairFunEq = prove (Term
`!(M:'a#'b->'c) N. (!p:'a#'b. M p = N p) =
                   (!(q:'a) (r:'b). M (q, r) = N (q, r))`,
REPEAT GEN_TAC THEN EQ_TAC THENL
[REPEAT STRIP_TAC THEN AR
,REPEAT STRIP_TAC
 THEN CONV_TAC (BINOP_CONV (RAND_CONV (REWR_CONV (GSYM PAIR)))) THEN AR]);

(* XP_FUN_EQ_CONV (t1, t2) (f = g) =
         |- (f = g) = (!t1 t2. f (t1, t2) = g (t1, t2)) *)

fun XP_FUN_EQ_CONV (t1, t2) =
 FUN_EQ_CONV THENC REWR_CONV pairFunEq
 THENC RAND_CONV (ABS_CONV (GEN_ALPHA_CONV t2))
 THENC GEN_ALPHA_CONV t1;

(* Define X_UNSKOLEM_CONV, a conversion inverse to X_SKOLEM_CONV, for
 pushing an existentially quantified function variable f inside
 universal quantifications over variables x1, ... xn of its (Curried)
```

```
 argument types, changing it into an existentially quantified variable,
 y say, (supplied as first argument to X_UNSKOLEM_CONV) of the result
 type of f, and replacing applications f x1 ... xn in the body with
 occurrences of y. Stops pushing when the type of f x1 ... xi
 equals that of y. *)

fun X_UNSKOLEM_CONV y t =
 let fun is_funtype ty =
       not (is_vartype ty) andalso (#Tyop (Rsyntax.dest_type ty) = "fun");
     fun argtype ty = hd (#Args (Rsyntax.dest_type ty));
     fun unskolemize (ptl_apn, tm) =
         if type_of ptl_apn = type_of y
         then Rsyntax.mk_exists {Bvar= y, Body= subst [ptl_apn |-> y] tm}
         else if is_funtype (type_of ptl_apn) andalso is_forall tm andalso
                 type_of (bvar (rand tm)) = argtype (type_of ptl_apn)
         then Rsyntax.mk_forall {Bvar= bvar (rand tm),
                 Body= unskolemize
                 (Rsyntax.mk_comb {Rator= ptl_apn, Rand= bvar (rand tm)},
                  body (rand tm))}
         else raise (HOL_ERR {message= "cannot continue to unskolemize",
                                  origin_function= "X_UNSKOLEM_CONV",
                                  origin_structure= "ante_allTacs"});
     val {Bvar= fvar, Body= uterm} = Rsyntax.dest_exists t
 in SYM (X_SKOLEM_CONV fvar (unskolemize (fvar, uterm))) end;

(* Rule, RESQ_INST: thm -> thm -> thm, a variant of RESQ_SPEC.

   |- !x::P.A
------------------ RESQ_INST (|- P t)
   |- A[t/x]                          (INST short for "instantiate") *)

fun RESQ_INST tisP =
 let val Pt = concl tisP;
     val t = rand Pt
 in fn allxPdotA => MP (DISCH Pt (RESQ_SPEC t allxPdotA)) tisP end;

(* Version of UNDISCH for an antecedent which may be T or a conjunction or
   an instance of REFL: idea is keep the world safe for ASM_REWRITE_TAC.*)

fun CONJ_UNDISCH th = (* now (11/15/99) with beta-conversion *)
 let val thm = CONV_RULE (LAND_CONV (DEPTH_CONV BETA_CONV)) th;
     val {ant=ante, ...} = Rsyntax.dest_imp (concl thm)
 in if ante = --`T`-- orelse (is_eq ante andalso
       (#lhs (Rsyntax.dest_eq ante) = #rhs (Rsyntax.dest_eq ante)))
    then CONV_RULE (LAND_CONV (REWRITE_CONV [])
              THENC REWR_CONV (CONJUNCT1 (SPEC_ALL IMP_CLAUSES))) thm
    else if is_conj ante then
              CONJ_UNDISCH (CONJ_UNDISCH (CONV_RULE ANTE_CONJ_CONV thm))
    else UNDISCH thm
 end;

fun CONJ_ASSUME_TAC th = (* similar; now with better beta-conversion *)
 let val thm = CONV_RULE (DEPTH_CONV BETA_CONV) th
 in if concl thm = --`T`-- orelse (is_eq (concl thm)
       andalso (#lhs (Rsyntax.dest_eq (concl thm)) =
               #rhs (Rsyntax.dest_eq (concl thm))))
    then ALL_TAC
    else if is_conj (concl thm) then
          let val (th1, th2) = CONJ_PAIR thm
          in CONJ_ASSUME_TAC th1 THEN CONJ_ASSUME_TAC th2 end
    else ASSUME_TAC thm end;

(* Note that CONJ_DISCH_TAC may fail, and so is REPEATable; where a never-
 failing version is needed, just use TRY CONJ_DISCH_TAC. *)

val CONJ_DISCH_TAC = DISCH_THEN CONJ_ASSUME_TAC;
```

```
val HALF_CONJ_DISCH_TAC = HALF_DISCH_THEN CONJ_ASSUME_TAC;

(* A tactic, EXISTS_UNIQUE_TAC, allowing to provide a witness in order
   to solve a unique existance (?!x. ...) goal. Example (in the presence
   of arithmeticTheory):

g`?!z:num. !w. z <= w`;            Initial goal: ?!z. !w. z <= w
e(EXISTS_UNIQUE_TAC (Term `0`));
 2 subgoals: [!w. 0 <= w] ?- !z. (!w. z <= w) ==> (z = 0)
                             !w. 0 <= w                    *)

fun EXISTS_UNIQUE_TAC u = CONV_TAC EXISTS_UNIQUE_CONV THEN
 (fn (g as (_, exanduq)) =>
  let val {conj1= ex, conj2= _} = Rsyntax.dest_conj exanduq;
      val {Bvar= x, Body= tx} = Rsyntax.dest_exists ex;
  in (CONV_TAC LEFT_AND_EXISTS_CONV
      THEN EXISTS_TAC u
      THEN CUMUL_CONJ_TAC THENL
      [ALL_TAC,
       DISCH_THEN (fn tu =>
        CONJ_ASSUME_TAC tu
        THEN SUBGOAL_THEN (Term`!(^x).(^tx ==> (^x = ^u))`)
        (fn allu => GEN_TAC THEN GEN_TAC
         THEN HALF_DISCH_THEN (fn tyth =>
          DISCH_THEN (fn tzth => ACCEPT_TAC (TRANS
           (MATCH_MP allu tyth) (SYM (MATCH_MP allu tzth))))))))]) g end);

(* An analogue of EXISTS_UNIQUE_TAC that works like STRIP_TAC on a
  unique existence top-level hypothesis.

                  A ?- (?!x. Q[x]) ==> M
           ==================================== STRIP_EXISTS_UNIQUE_TAC
           A, Q[x] ?- (!y. Q[y] ==> (y = x)) ==> M

  Normally STRIP_EXISTS_UNIQUE_TAC will be followed immediately by
  DISCH_TAC, but this step has intentionally not been included in case
  some other disposition of the uniqueness hypothesis is preferred.

  Example: g`(?!z:num. !w. z <= w) ==> M`; e(STRIP_EXISTS_UNIQUE_TAC);
  results in:  (!z'. (!w. z' <= w) ==> (z' = z)) ==> M
               ------------------------------------
                  !w. z <= w                                  *)

val STRIP_EXISTS_UNIQUE_TAC =
 CONV_TAC ((LAND_CONV EXISTS_UNIQUE_CONV) THENC ANTE_CONJ_CONV
           THENC LEFT_IMP_EXISTS_CONV)
 THEN (fn (g as (_, uimp)) =>
       let val {Bvar= x, Body= _} = Rsyntax.dest_forall uimp
       in (GEN_TAC
           THEN DISCH_THEN (fn wit => CONJ_ASSUME_TAC wit THEN
                 DISCH_THEN (fn cimp => MP_TAC (symOfEqn
                  (MATCH_MP (CONV_RULE (ONCE_DEPTH_CONV ANTE_CONJ_CONV)
                             cimp) wit))))) g end);

(* Following two thm-tacticals, IMP_RES_THENL and IMP_RES_ASM_THENL,
   iterate IMP_RES_THEN for a fixed number of stages (so that the
   theorem argument should have that many iterated hypotheses at least)
   and rewrite the theorem(s) emerging from each stage; like IMP_RES_THEN,
   the disposition of the finally emerging theorem(s) is for the th-tactic
   argument to determine. An additional, first, theorem list list argument
   has a sublist for every stage of IMP_RES_THEN to be applied, and each
   sublist gives theorems to be used in rewriting each conclusion at that
   stage; for IMP_RES_ASM_THENL, the current assumptions are thrown in
   to every stage of rewriting as well. See fstbi_natTh for an example. *)

fun IMP_RES_THENL [] thtac = thtac
  | IMP_RES_THENL (ths :: thss) thtac =
```

```
     IMP_RES_THEN (fn th' =>
      IMP_RES_THENL thss thtac (REWRITE_RULE ths th'));

fun IMP_RES_ASM_THENL [] thtac = thtac
  | IMP_RES_ASM_THENL (ths :: thss) thtac =
     IMP_RES_THEN (fn th' => ASSUM_LIST (fn asms =>
       IMP_RES_ASM_THENL thss thtac (REWRITE_RULE (ths @ asms) th')));

(* What would be the specification for RES_ABSTRACT if pred_set's had
 not been put in in place of plain predicates: *)

val RES_ABSTRACT_PRED = REWRITE_RULE [SPECIFICATION] RES_ABSTRACT;

(* RES_ABSTRACT_PRED = |- !p m x. p x ==> (RES_ABSTRACT p m x = m x)  *)

(* The old RESQ_FORALL_CONV, putting in a predication rather than an IN
 in the hypothesis: |- !x::P. M = !x. P x ==> M *)

val RESQ_FORALL_CONV = RES_FORALL_CONV THENC
 RAND_CONV (ABS_CONV (LAND_CONV (REWR_CONV SPECIFICATION)));

(* following will reduce restricted beta-redexes, incurring an assumption;
 RES_BETA_CONV gives new Hurd-style (with IN); RESQ_BETA_CONV gives
 old (with predication) style assumption. *)

fun RESQ_BETA_CONV app =
 let val rab = rator app;
     val (pred, func) = (rand (rator rab), rand rab);
     val imp = ISPECL [pred, func, rand app] RES_ABSTRACT_PRED
 in (REWR_CONV (UNDISCH imp) THENC BETA_CONV) app end;

fun RES_BETA_CONV app =
 let val rab = rator app;
     val (pred, func) = (rand (rator rab), rand rab);
     val imp = ISPECL [pred, func, rand app] RES_ABSTRACT
 in (REWR_CONV (UNDISCH imp) THENC BETA_CONV) app end;

(* abs_remove turns equations to lambda-abstractions into universally
 quantified equations with application to the variables on the left; in
 two versions to use the two flavors of RES(Q)_BETA_CONV. pure_abs_remove
 is for when one wants to see IN's in the hypotheses of the result. *)

fun gen_abs_remove Pure th =
let fun abs_rem th =
 if is_forall (concl th)
 then let val (v, th') = SPEC_VAR th
      in GEN v (abs_rem th') end
 else if is_imp (concl th)
 then let val {ant= a, conseq= c} = Rsyntax.dest_imp (concl th)
      in DISCH a (abs_rem (UNDISCH th)) end
 else if is_let (concl th)
 then abs_rem (CONV_RULE let_CONV th)
 else if is_conj (concl th)
 then CONJ (abs_rem (CONJUNCT1 th)) (abs_rem (CONJUNCT2 th))
 else if is_eq (concl th)
 then let val {lhs= header, rhs= defn} = Rsyntax.dest_eq (concl th)
      in if is_abs defn
         then let val v = #Bvar (Rsyntax.dest_abs defn)
              in abs_rem (CONV_RULE
                              (X_FUN_EQ_CONV v THENC
                     RAND_CONV (ABS_CONV (RAND_CONV BETA_CONV))) th) end
         else if is_pabs defn
         then let val th' = abs_rem (CONV_RULE
                           (LAND_CONV (REWR_CONV (GSYM UNCURRY_CURRY_THM))
                            THENC REWR_CONV UNCURRY_ONE_ONE_THM) th);
              in CONV_RULE (ONCE_DEPTH_CONV (REWR_CONV CURRY_DEF)) th' end
         else if is_res_abstract defn
```

```
        then
         let val (v, P, _) = dest_res_abstract defn;
             val Pv = if Pure then Term`^v IN ^P`
                      else Rsyntax.mk_comb {Rator= P, Rand= v};
             val th' = SPEC_ALL (CONV_RULE (X_FUN_EQ_CONV v) th);
             val th'' = UNDISCH (DISCH Pv th');
             val th''' = CONV_RULE (RAND_CONV
                          (if Pure then RES_BETA_CONV
                                   else RESQ_BETA_CONV)) th'';
             val thiv = DISCH Pv (abs_rem th''')
         in GEN v (if is_abs P andalso not Pure
                   then CONV_RULE (LAND_CONV BETA_CONV) thiv else thiv)
 end else th end else th
in abs_rem th end;

val pure_abs_remove = gen_abs_remove true;

val abs_remove = gen_abs_remove false;

val let_remove = CONV_RULE (DEPTH_CONV let_CONV); (* sometimes useful *)

end;
```

D.8 ante_allTacs.sig

```
(* File: ante_allTacs.sig, created 7/27/02 *)
(* Author: F. Lockwood Morris  <lockwood@ecs.syr.edu> *)

signature ante_allTacs =
sig

type tactic = Abbrev.tactic;
type thm_tactic = Abbrev.thm_tactic;
type conv = Abbrev.conv;
type thm = Thm.thm;
type term = Term.term

val CUMUL_CONJ_TAC : tactic;
val T_TAC : tactic;
val TR_TAC : tactic;
val ANTE_DROP_TAC : tactic;
val UNASSUME_TAC : thm -> tactic;
val ulist : 'a -> 'a list;
val HALF_DISCH_THEN : thm_tactic -> tactic;
val SWAP_HALF_DISCH_THEN : thm_tactic -> tactic;
val HALF_DISCH_TAC : tactic;
val SWAP_HALF_DISCH_TAC : tactic;
val LLAND_CONV : conv -> conv;
val LRAND_CONV : conv -> conv;
val RLAND_CONV : conv -> conv;
val RRAND_CONV : conv -> conv;
val ASM_SYM_TAC : term -> tactic;
val AR : tactic;
val LEMMA_MP_TAC : term -> tactic;
val LEMMA_TAC : term -> tactic;
val MATCH_MP2 : thm -> thm -> thm -> thm;
val EQ_HYP_TAC : term -> tactic;
val lines : string -> term -> bool;
val asm : string -> term -> bool;
val with_asm : string -> thm_tactic -> tactic;
```

```
val drop_asm : string -> tactic;
val und_asm: string -> tactic;
val tty : unit -> bool;
val is_unap : string -> term -> bool;
val is_binap : string -> term -> bool;
val REV_EXISTS_CONV : conv;
val REV_FORALL_CONV : conv;
val DUP_TAC : 'a -> 'a list;
val DUP_ARGS_TAC : 'a * 'a -> ('a -> 'b) -> 'b list;
val FORALL_IMP_LIST_RULE : (thm -> thm list) -> thm -> thm list;
val GCONJ_LIST : int -> thm -> thm list;
val impOfEqn : thm -> thm;
val impByOfEqn : thm -> thm;
val symOfEqn : thm -> thm;
val GCONJ_CONV : term -> thm;
val GCONJUNCTS : thm -> thm list;
val GCONJ_PAIR : thm -> thm * thm;
val GCONJ_TRIP : thm -> thm * thm * thm;
val COND_RW_TAC : thm -> tactic;
val THENSGS : tactic * tactic -> tactic;
val THENFIN : tactic * tactic -> tactic;
val CRE : thm_tactic;
val CR : thm_tactic;
val CREL : thm list -> tactic;
val CRL : thm list -> tactic;
val GEXT : thm -> thm;
val XL_FUN_EQ_CONV : term list -> conv;
val XL_FUN_EQ_TAC : term list -> tactic;
val XP_FUN_EQ_CONV : term * term -> conv;
val X_UNSKOLEM_CONV : term -> conv;
val RESQ_INST : thm -> thm -> thm;
val CONJ_UNDISCH : thm -> thm
val CONJ_ASSUME_TAC : thm -> tactic;
val CONJ_DISCH_TAC : tactic;
val HALF_CONJ_DISCH_TAC : tactic;
val EXISTS_UNIQUE_TAC : term -> tactic;
val STRIP_EXISTS_UNIQUE_TAC : tactic;
val IMP_RES_THENL : thm list list -> thm_tactic -> thm_tactic;
val IMP_RES_ASM_THENL : thm list list -> thm_tactic -> thm_tactic;
val RES_ABSTRACT_PRED : thm;
val RESQ_FORALL_CONV : conv;
val RESQ_BETA_CONV : conv;
val RES_BETA_CONV : conv;
val abs_remove : thm -> thm;
val pure_abs_remove : thm -> thm;
val let_remove : thm -> thm;

end;
```

Bibliography

[Bib75] K. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, June 1975.

[BL73] D. E. Bell and L. J. La Padula. Secure computer systems: Mathematical foundations. Technical Report Technical Report MTR-2547, Vol. I, MITRE Corporation, Bedford, MA, March 1973.

[CMOV10] Shiu-Kai Chin, Sarah Muccio, Susan Older, and Thomas N. J. Vestal. Policy-based design and verification for mission assurance. In Igor V. Kotenko and Victor A. Skormin, editors, *MMM-ACNS*, volume 6258 of *Lecture Notes in Computer Science*, pages 125–138. Springer, 2010.

[CO11] Shiu-Kai Chin and Susan Older. *Access Control, Security, and Trust: A Logical Approach*. CRC Press, 2011.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[HOLa] *The HOL System Description*. http://hol.sourceforge.net/documentation.html.

[HOLb] *The HOL System Logic*. http://hol.sourceforge.net/documentation.html.

[HOLc] *The HOL System Reference*. http://hol.sourceforge.net/documentation.html.

[HOLd] *The HOL System Tutorial*. http://hol.sourceforge.net/documentation.html.

[OC12] Susan Older and Shiu-Kai Chin. Engineering assurance at the undergraduate level. *IEEE Security & Privacy*, 10(6):74–77, November/December 2012.