

# Certified Security by Design Using Higher Order Logic<sup>1</sup>

Shiu-Kai Chin and Susan Older  
Department of Electrical Engineering and Computer Science  
Syracuse University, Syracuse, New York 13244

Fall 2016  
Version 1.4 (August 2016)

<sup>1</sup>The development of this book was partially supported by the US National Science Foundation (Award #1245867).

Copyright © 2016 by Shiu-Kai Chin

All rights reserved. No part of this publication may be reproduced, stored, or transmitted in any form  
without permission of the author

*In memory of Lockwood Morris for his unwavering support  
To the engineers and computer scientists who design and deliver the systems on which we depend*



---

# Contents

---

<b>Preface</b>	<b>15</b>
<b>I Introduction and Motivation</b>	<b>17</b>
<b>1 Introduction</b>	<b>19</b>
1.1 What this book is about . . . . .	19
1.2 Our audience . . . . .	20
1.3 What computer engineers and scientists need to know . . . . .	21
1.4 Certified Security by Design . . . . .	22
1.4.1 Concepts of Operation . . . . .	22
1.4.2 A motivating example . . . . .	24
1.4.3 What you will learn . . . . .	25
<b>II Lab Exercises: Introduction to ML</b>	<b>27</b>
<b>2 Getting Started with Linux, ML, and Emacs</b>	<b>29</b>
2.1 Getting set up . . . . .	29
2.2 Looking around using Linux commands within a terminal . . . . .	30
2.3 Starting HOL and Evaluating ML Expressions . . . . .	31
2.4 Starting Emacs . . . . .	32
2.5 Exercises . . . . .	34
<b>3 An Introduction to ML within HOL</b>	<b>37</b>
3.1 Values and Types in ML . . . . .	38
3.2 Compound Values . . . . .	42
3.3 Value Declarations . . . . .	44
3.4 Exercises . . . . .	49
<b>4 Functions in ML</b>	<b>51</b>
4.1 Functions, Evaluation Rules, and Values . . . . .	51
4.2 Functions, Evaluation Rules and Values in ML . . . . .	54
4.3 Naming Functions in ML . . . . .	55
4.4 Defining Functions Using Pattern Matching in ML . . . . .	57
4.5 Let Expressions in ML . . . . .	59
4.6 Exercises . . . . .	62

<b>5</b>	<b>Polymorphism and Higher-Order Functions in ML</b>	<b>63</b>
5.1	Polymorphism . . . . .	64
5.2	Higher-Order Functions . . . . .	66
5.3	Exercises . . . . .	69
<b>III</b>	<b>Lab Exercises: Introduction to HOL</b>	<b>71</b>
<b>6</b>	<b>Introduction to HOL</b>	<b>73</b>
6.1	HOL Terms . . . . .	73
6.1.1	Entering HOL terms and types . . . . .	73
6.2	Exercises . . . . .	75
<b>7</b>	<b>Constructing and Deconstructing HOL Terms</b>	<b>77</b>
7.1	Introduction to Constructing and Deconstructing HOL Terms . . . . .	77
7.2	Manipulating HOL terms . . . . .	79
7.3	Exercises . . . . .	82
<b>8</b>	<b>Introduction to Forward Proofs in HOL</b>	<b>83</b>
8.1	Structure of HOL Theorems . . . . .	83
8.2	Inference Rules in HOL . . . . .	85
8.3	Relating Proofs in HOL to Traditional Proofs . . . . .	86
8.4	Exercises . . . . .	89
<b>9</b>	<b>Goal-Oriented Proofs</b>	<b>91</b>
9.1	Goal-Oriented Proofs . . . . .	91
9.1.1	Basic Techniques . . . . .	91
9.1.2	Tactics . . . . .	92
9.2	Using Tactics and Tacticals . . . . .	94
9.2.1	Tactics . . . . .	94
9.2.2	Tacticals . . . . .	96
9.2.3	Working with Assumptions . . . . .	100
9.3	Using HOL within Emacs . . . . .	101
9.4	High-Level Proof Tactics . . . . .	106
9.5	Exercises . . . . .	107
<b>10</b>	<b>Dealing with Assumptions in Goal-Oriented Proofs</b>	<b>109</b>
10.1	Rewriting and Resolution . . . . .	109
10.2	Manipulating the Assumptions with PAT_ASSUM . . . . .	112
10.3	More Examples . . . . .	114
10.4	Exercises . . . . .	126
<b>11</b>	<b>Defining Theories, Types, and Functions in HOL</b>	<b>127</b>
11.1	Defining New Theories in HOL . . . . .	127
11.2	Using Existing Theories and Types in HOL . . . . .	129
11.3	Adding New Definitions to HOL . . . . .	130
11.4	Proofs Using Structural Induction . . . . .	132
11.5	Defining New Types and Their Properties in HOL . . . . .	135

11.6 Exercises . . . . .	141
<b>12 Inductive Relations in HOL</b>	<b>143</b>
12.1 Inductive Definitions in HOL . . . . .	143
12.2 Reasoning About Inductive Relations . . . . .	149
12.2.1 Even_rules . . . . .	149
12.2.2 Odd_rules . . . . .	152
12.2.3 not_odd_0 . . . . .	152
12.2.4 not_even_1 . . . . .	154
12.2.5 even_not_odd . . . . .	155
12.2.6 odd_not_even . . . . .	158
12.2.7 even_theorem . . . . .	159
12.2.8 odd_theorem . . . . .	162
12.2.9 not_even_lemma . . . . .	162
12.2.10 not_odd_lemma . . . . .	164
12.2.11 even_odd_theorem . . . . .	164
12.2.12 odd_even_theorem . . . . .	165
12.2.13 evenEven_lemma . . . . .	165
12.2.14 oddOdd_lemma . . . . .	167
12.2.15 Even_even_Odd_odd_lemma . . . . .	167
12.2.16 even_is_Even . . . . .	168
12.2.17 odd_is_Odd . . . . .	169
12.2.18 even_is_EVEN . . . . .	169
12.2.19 Even_is_EVEN . . . . .	175
12.2.20 odd_is_ODD . . . . .	175
12.2.21 Odd_is_ODD . . . . .	175
12.3 Exercises . . . . .	176
<b>IV Lab Exercises: Access-Control Logic in HOL</b>	<b>179</b>
<b>13 An Access-Control Logic in HOL</b>	<b>181</b>
13.1 Syntax . . . . .	181
13.2 Semantics . . . . .	182
13.3 Inference Rules . . . . .	183
13.4 Describing Access-Control Concepts in the C2 Calculus . . . . .	184
13.5 The Access-Control Logic in HOL . . . . .	186
13.6 Syntax of the Access-Control Logic in HOL . . . . .	187
13.7 Semantics of the Access-Control Logic in HOL . . . . .	188
13.8 C2 Inference Rules in HOL . . . . .	190
13.9 Using the Access Control Logic in HOL . . . . .	190
13.9.1 Specifying Directory Paths for HOL . . . . .	190
13.9.2 Documentation for the ACL Implementation in HOL . . . . .	191
13.9.3 The ACL_ASSUM and CONTROLS Inference Rules . . . . .	193
13.9.4 Using the ACL_ASSUM and Controls Inference Rules . . . . .	195
13.9.5 Goal-Oriented Access-Control Logic Proofs . . . . .	199
13.9.6 More Inference Rules and Tactics . . . . .	206

13.10 Exercises . . . . .	211
<b>14 Concepts of Operation in the Access-Control Logic and HOL</b>	<b>215</b>
14.1 Concepts of Operations . . . . .	215
14.2 A Command and Control Example . . . . .	216
14.3 Verifying the Command and Control Example in HOL . . . . .	218
14.4 Exercises . . . . .	222
<b>V Lab Exercises: Cryptographic Components</b>	<b>225</b>
<b>15 Cryptographic Operations in HOL</b>	<b>227</b>
15.1 Properties, Reality, Purposes, and Models . . . . .	227
15.2 Building and Loading Cipher Theory . . . . .	228
15.3 An Algebraic Model of Symmetric Key Encryption in HOL . . . . .	229
15.3.1 Idealized Behavior . . . . .	230
15.3.2 Modeling Idealized Behavior in HOL . . . . .	230
15.4 Cryptographic Hash Functions . . . . .	232
15.5 Asymmetric-Key Cryptography . . . . .	233
15.5.1 Digital Signatures . . . . .	235
15.6 Exercises . . . . .	237
<b>VI Lab Exercises: Transition Systems</b>	<b>239</b>
<b>16 High-Level State Machines</b>	<b>241</b>
16.1 State Machines . . . . .	241
16.1.1 Defining Parameterized State Machines in HOL . . . . .	242
16.1.2 Defining State Machines Using Configurations . . . . .	243
16.1.3 Proving Equivalence Properties of $TR\ x$ . . . . .	245
16.1.4 Proving $TR\ x$ is Deterministic . . . . .	252
16.1.5 Proving $TR\ x$ is Completely Specified . . . . .	258
16.1.6 Proving the Equivalence of $Trans\ x$ and $TR\ x$ . . . . .	261
16.1.7 Defining Specialized Inference Rules for $smTheory$ . . . . .	262
16.2 Defining State Machines Using $smTheory$ . . . . .	263
16.2.1 Defining Machine $M_0$ Using $smTheory$ . . . . .	264
16.2.2 Defining a Simple Thermostat . . . . .	269
16.3 Exercises . . . . .	273
<b>17 Secure State Machines</b>	<b>277</b>
17.1 A High-Level Secure State-Machine . . . . .	277
17.1.1 Building and Loading $ssm1Theory$ . . . . .	278
17.1.2 Basic Types Used by $ssm1Theory$ . . . . .	279
17.1.3 Interpreting <i>configurations</i> in the Access-Control Logic . . . . .	280
17.1.4 Defining the Configuration Transition Relation $TR$ . . . . .	281
17.1.5 Equality Theorems for $TR$ . . . . .	283
17.2 Defining Secure State-Machine $SM_0$ Using $ssm1Theory$ . . . . .	285
17.2.1 $SM_0$ Datatypes . . . . .	285



17.2.2	$SM_0$ Next-State Function . . . . .	287
17.2.3	$SM_0$ Next-Output Function . . . . .	287
17.2.4	Input Authentication Function . . . . .	288
17.2.5	Certificates Establishing Security Context . . . . .	289
17.2.6	Proof that Carol's Commands are Discarded . . . . .	289
17.2.7	Proof that Alice's Commands are Completely Mediated . . . . .	291
17.3	Exercises . . . . .	303
17.4	Source Files . . . . .	308
17.4.1	Holmakefile . . . . .	308
17.4.2	satListScript.sml . . . . .	308
17.4.3	ssminfRules.sml . . . . .	309
17.4.4	ssminfRules.sig . . . . .	310
17.4.5	ssm1Script.sml . . . . .	310
17.4.6	SM0Script.sml . . . . .	315
<b>18</b>	<b>Secure State Machine Refinements</b>	<b>329</b>
<b>VII</b>	<b>Appendix</b>	<b>331</b>
<b>19</b>	<b>Using Holmake</b>	<b>333</b>
19.1	A Simple Example . . . . .	333
19.2	Using Theories and Libraries in Other Subdirectories . . . . .	335
<b>20</b>	<b>Documentation for ML and HOL</b>	<b>337</b>
20.1	Documentation for Moscow ML . . . . .	337
20.2	Documentation for HOL . . . . .	337
<b>21</b>	<b>Summary of the Access-Control Logic</b>	<b>341</b>
21.1	Syntax . . . . .	341
21.2	Core Rules, Derived Rules, and Extensions . . . . .	342
	<b>Bibliography</b>	<b>347</b>



---

# List of Tables

---

6.1	HOL Notation for Higher Order Logic Terms . . . . .	76
7.1	Predicates, Deconstructors, and Constructors for HOL Terms . . . . .	78
9.1	Proof Management Functions . . . . .	94



---

## List of Figures

---

1.1	Virtuous Cycle of Assurance: Specify, Design, and Verify . . . . .	20
1.2	Graphical Specification of Behavior . . . . .	20
1.3	Flow of Command and Control (C2) for a Simple CONOPS . . . . .	23
1.4	A Networked Thermostat and Its Operating Environment . . . . .	24
8.1	HOL Inference Rules . . . . .	87
9.1	Tactics . . . . .	96
9.2	Tactic Proof Tree . . . . .	97
13.1	CONOPS Statements and Their Representation in the C2 Calculus . . . . .	182
13.2	Kripke Semantics of Access-Control Logic Formulas . . . . .	182
13.3	Inference rules for the access-control logic . . . . .	183
13.4	Access-Control Logic Syntax in HOL . . . . .	187
13.5	Syntax of Principal Expressions, Integrity and Security Labels, and Kripke Structures in HOL	188
13.6	C2 Formulas and Their Representation in HOL . . . . .	188
13.7	HOL Theorems Corresponding to C2 Calculus Inference Rules (1 of 2) . . . . .	191
13.8	HOL Theorems Corresponding to C2 Calculus Inference Rules (2 of 2) . . . . .	192
14.1	Flow of Command and Control (C2) for a Simple CONOPS . . . . .	216
14.2	A Command and Control Chain . . . . .	217
14.3	Proof Justifying Operator's Action . . . . .	218
14.4	Proof Justifying Application's Action . . . . .	218
14.5	CONOPS Using Keys and Certificate Authorities . . . . .	223
15.1	Symmetric-Key Encryption and Decryption . . . . .	229
15.2	Option Theory in HOL . . . . .	230
15.3	Definitions and Properties of Symmetric Encryption and Decryption . . . . .	231
15.4	Definition of Digests and their Properties . . . . .	232
15.5	Asymmetric-Key Encryption and Decryption . . . . .	233
15.6	Definitions and Properties of Asymmetric Keys and Messages . . . . .	233
15.7	Definitions and Properties of Asymmetric Decryption . . . . .	234
15.8	One-to-One Properties of Asymmetric Decryption . . . . .	235
15.9	Digital Signature Generation . . . . .	235
15.10	Digital Signature Verification . . . . .	236
15.11	Digital Signature Generation, Verification, and Their Properties . . . . .	237
16.1	Parameterized State-Transition Relation . . . . .	242
16.2	Input and Output Streams . . . . .	243

---

16.3	Finite-State Machine Example $M_0$ . . . . .	264
16.4	Tabular Description of a Simple Thermostat . . . . .	269
16.5	Machine $M_1$ . . . . .	273
16.7	Partial State-Transition Diagram for Down Counter . . . . .	275
16.6	Tabular Description of Down-Counter Behavior . . . . .	275
19.1	Example Theory Script . . . . .	336
20.1	HOL Reference Page . . . . .	338
21.1	Summary of core rules for the access-control logic . . . . .	343
21.2	Summary of useful derived rules . . . . .	344
21.3	Summary of rules for delegation . . . . .	344
21.4	Inference rules for relating security levels . . . . .	345
21.5	Inference rules for relating integrity levels . . . . .	345
21.6	Logical rules regarding principal equality . . . . .	345

---

# Preface

---

Our purpose in writing this textbook is to serve the needs of engineers and computer scientists who are responsible for designing, implementing, and verifying secure computer and information systems. This purpose is unchanged from that of our previous textbook, *Access Control, Security, and Trust: A Logical Approach*, [Chin and Older, 2010]. As before, our methods are based on the application of logic as a means for describing, reasoning about, and verifying the properties of systems. We use logic from the conceptualization stage, through the design phase, and up to and including verification and certification. Our intent is to make this material accessible to upper division undergraduate students in computer science and engineering. What we present here has been tried out on undergraduate students from many different undergraduate colleges and universities.

This text book builds upon our previous work on access control and adds two more dimensions: (a) transition systems, and (b) formal verification computer-assisted reasoning tools. Why these additions?

First, all but the simplest combinational logic functions are transition systems, i.e., they have some notion of state. The logic of transition systems is useful to describe both the behavior and properties of systems. When combined with the multi-agent modal logic that is the access-control logic used for describing and reasoning about the elements of access-control decisions, we are able to describe the behavior of systems over time while accounting for the security policies affecting system operations.

Second, assurance of correctness matters. No matter how simple a calculation or proof might be, the possibility of human error is ever present. Just as the use of computer-aided design (CAD) tools is essential for producing microprocessors with billions of transistors on a single chip, so is the use of computer-assisted reasoning tools, such as theorem provers, necessary to formally verify assurance arguments. The use of reasoning tools such as theorem provers provide a degree of confidence unmatched by less rigorous means depending on either pencil and paper calculations or simulations that often are incomplete case analyses of a system. *These tools are an effective and essential antidote to self delusion.*

CAD tools and programming languages sometimes change and evolve quickly. The particular theorem prover we use is the the Cambridge University Higher Order Logic (HOL) theorem prover. There are other theorem provers and our use of HOL is not intended as an implicit judgment of its superiority over the rest. Rather, it reflects HOL's longevity, openness, and extensive libraries of theories developed since its inception in the early 1980s. To the extent possible, we have tried to minimize the dating this book by focusing on the long-lasting parts of HOL.

As with our previous book, we developed much of the content of this book in our own courses at Syracuse University and for the Air Force Research Laboratory Information Directorate's Information Assurance Research internships. Our students benefited from having illustrations and exercises to learn from and gain deeper insights. Thus, we have included numerous examples to illustrate principles, as well as many exercises to serve as assessments of knowledge.

**Acknowledgments** We are grateful for the support of numerous colleagues and students. In particular, we are grateful for the partnership we have enjoyed with the Air Force Research Laboratory since 2003. The partnership has allowed us access to young and fresh minds who are tomorrow's leaders. In memorium, we

are deeply grateful to Lockwood Morris. Lockwood was one of the creators of ML, a contributor to HOL, a good colleague, and always willing to help. The access-control logic in HOL would not have been possible without his help.



## **Part I**

# **Introduction and Motivation**



# Introduction

---

## 1.1 What this book is about

This book is about assurance in cyberspace. When we say cyberspace, we mean a realm in which command, control, communication, and information exist within networks of computers and sensors. Actors in cyberspace include people, machines, and organizations. Systems are often comprised of all of the above. Systems are typically people, computers, sensors, machines, and networks organized to fulfill a purpose, accomplish a mission, or supply a service. Our society functions with people and things interacting using networks of sensors and computer-enabled command and control (C2).

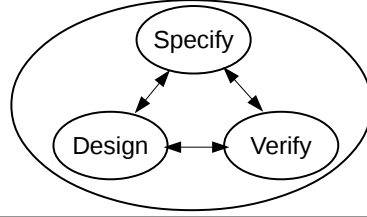
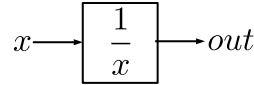
Our focus is on computer-enabled C2. Our interest is motivated by the increasing use of intelligent sensors, controllers, and computers in common objects. These objects include cars, smart phones, medical devices, and SCADA (supervisory control and data acquisition) devices controlling the valves, relays, and switches in our water, power, and telecommunications infrastructure. In cars, physically isolated mechanical devices once controlled the locks, brakes, and fuel injection. Now, all these functions are controlled by special-purpose digital controllers and computers networked together and accessible from the Internet or wireless devices.

Networked computer-enabled C2 is convenient and vulnerable. Networked C2 enables instantaneous transfer of millions of dollars among commercial banks, remote control of unmanned aerial vehicles (UAVs), and homeowners to control their furnaces and lock their doors while away from home. Networked C2 also makes possible electronic theft of funds, hijacking of UAVs, and home invasion. Increasingly, the safety and security of society depends on the safety and security of networked C2. The security and integrity of networked C2 must be *assured* by design, as opposed by afterthought.

When we say that a system is assured to be secure and will act with integrity, we are making a promise that a system will in fact be secure and have integrity. What must we do to enable people to have confidence in our promises? Confidence in a system, such as a commercial airliner or passenger car, arises from a justified belief that it was manufactured properly, tested, and verified to have met rigorous safety and security standards. A combination of tests, simulations, and mathematical analyses are used to justify safety and security claims.

Simulation and testing without mathematical analyses are insufficient for guaranteeing or verifying that properties such as correctness, security, or integrity hold true for a system. Simulation and testing rely on a tester's ability to conceive of test cases. Given the virtually infinite number of potential test cases, relying solely on a finite number of test cases is folly. Testers devise tests for cases for which they can conceive. It is difficult to devise test cases for unconceived situations. Testing and simulation are inherently incomplete because only a finite number of test cases can be run.

Mathematical proofs are a means to address the deficiencies of testing and simulation because we can utilize techniques such as structural induction and algebraic properties to reason about an infinite number of cases. To the extent that our mathematical models accurately describe our systems, mathematical proofs provide a means to prove properties of mathematical models of systems over all cases of interest. Because they deal with all cases, as opposed to some cases, mathematical proofs provide greater assurance of security

**Figure 1.1** Virtuous Cycle of Assurance: Specify, Design, and Verify**Figure 1.2** Graphical Specification of Behavior

and integrity, when compared to testing and simulation alone. Proving a security property of a system's formal definition and description shows that the system as defined is inherently secure. This is one important means to show a system is secure by design.

Nevertheless, models are approximations of reality. Formal proofs of safety and security do not eliminate unexpected behavior in the real world. A significant benefit of formal proofs is the clarity and insight they provide into a system's behavior. When the unexpected happens, we can determine the root causes of problems and adapt more quickly than we might without such insights.

Typically, attempting to prove a property of a system uncovers unconsidered cases or scenarios. This additional knowledge results in further modifications and refinements to a design so that the specified or modified description of behavior is achieved. This cycle of designing a system with a specification in mind, attempting to prove the design has the specified behavior, and then refining or modifying the design or specification based on the attempted proof, is one example of what we call the *virtuous cycle of assurance*. Figure 1.1 illustrates the relationships among specifications, designs, and verifications or proofs. Each is related to the other. There is no magic. All relevant design details and properties must be defined and verified before we have rigorous and formal assurances of correctness and security.

### Example 1.1

Suppose you are given the graphical specification shown in Figure 1.2. We are tempted to say that for all values of  $x$ ,  $out = \frac{1}{x}$ . Of course, if one tries to prove this statement we quickly realize that this holds for all values of  $x$  *except* for  $x = 0$ . Once we remember that division is not defined when  $x = 0$ , we modify and refine our definitions and behavioral expectations to exclude the case when  $x = 0$ .

Suppose that after our design is verified and implemented it is used as a component in another system. Suppose, too, that this new system displays some anomalous behavior that is traced to our design. Because we proved for all values of  $x \neq 0$  that  $out = \frac{1}{x}$ , we immediately check to see if the anomalous behavior occurs when  $x = 0$ . We discover that the users of our design did not exclude the possibility that  $x = 0$  as an input case.  $\diamond$

## 1.2 Our audience

We wrote this book for people who specify, design, and verify computer and information systems that must be trustworthy and secure. As the bachelors of science degree in engineering and computer science defines

the baseline capabilities of the computer engineering and science professions, we wrote this book with both *undergraduate and graduate* students in mind. We have taught functional programming languages such as ML [Paulson, 1996], theorem provers such as HOL [Gordon and Melham, 1993], and modal logic specialized to reason about access control [Chin and Older, 2010], to both undergraduate and graduate students [Older and Chin, 2012, Chin, 2015, Older and Chin, 2002].

Most information systems or computers have security requirements. Common examples include computers handling sensitive information such as financial information, health records, or military secrets. Computer engineers and scientists are responsible for designing, building, testing, or certifying systems that have security concerns. These concerns possibly include the following questions.

- Who or what can access protected resources?
- How do we protect the confidentiality, integrity, and availability of information?
- Who or what is trusted or believed?
- What are the compelling reasons to conclude a system is worthy of being trusted?

For example, if you are responsible for the specification, design, or operation of computers holding bank accounts, you are very concerned about the following questions.

- Who can withdraw funds from a customer's bank account electronically?
- Who is allowed to alter the balance or available funds in a customer's account?
- Who has authority to grant account access?
- What evidence is there to substantiate that the computerized banking system is secure and operating correctly?

### 1.3 What computer engineers and scientists need to know

An oft-quoted and ignored security principle is *security must be designed into systems from the start*. Another oft-quoted and ignored design principle is complete mediation: *every access to every object must be checked for authority*, [Saltzer and Schroeder, 1975]. Systems designers routinely ignore the above two principles. Thus, the bedrock of security, knowing who has access to what and under what circumstances in terms of policies, concepts of operations, and enforcement mechanisms, is largely missing. This observation is not to say that designers and engineers are malicious or lazy as a group. Quite the contrary, designers and engineers as a group are dedicated to building systems correctly and securely. What they lack are the mathematical and logical tools to help them develop and verify their thinking and designs.

#### **Example 1.2**

As an experiment, try to think about something without using words or language. As another experiment, try describing an algebraic property or procedure without using mathematical symbols. The point of these mental experiments is to illustrate the difficulty of describing and reasoning about concepts without appropriate language, symbols, and properties. Security is no different. How can designers be held accountable for formally verifying access-control policies, concepts of operations, software, operating systems, and hardware unless they have the necessary technology in terms of languages, calculi, and computer-assisted reasoning tools to describe and verify their access control policies and enforcement mechanisms? ◇

Any engineer or computer scientist who has designed, certified, or worked with systems of any size or consequence knows that a key question is *how will we know?* They know that undetected design flaws or inappropriate assumptions that are built into deployed systems are potentially disastrous and life threatening. They know that flaws in deployed systems are orders of magnitude more costly to remedy when compared to corrections made in the design phase. They know that undetected flaws potentially destroy systems, destroy reputations, and destroy credibility, leading to failed missions, failed services, and failed corporations.

Experience shows that expert system designers are those people who combine their experience and intuition with mathematics and logic. Experience and intuition are powerful tools that inform the selection of a design approach. Mathematics and logic are unparalleled in providing assurances of correct coverage of all cases and instances, some of which might not have been imagined by designers or certifiers.

We follow the same approach used by civil, mechanical, and electrical engineers. Mathematics and logic properly used clarify the underlying principles and properties of systems. Systems with mathematical and logical descriptions are amenable to independent and automated verification and testing. The effects of system changes or consequences of altering assumptions are easier to deduce with logic than without.

Flaws and misconceptions often exist *between* levels of abstraction in systems. For example, how will we know that a security policy related to information integrity is correctly implemented by hardware and software together? Requirements writers, software engineers, and hardware engineers might interpret the meaning of integrity differently leading to improper assumptions, flawed policies, flawed designs, and failed systems. Our approach to dealing with this observation is to use a logic that spans many levels of abstraction including hardware, software, and policy.

## 1.4 Certified Security by Design

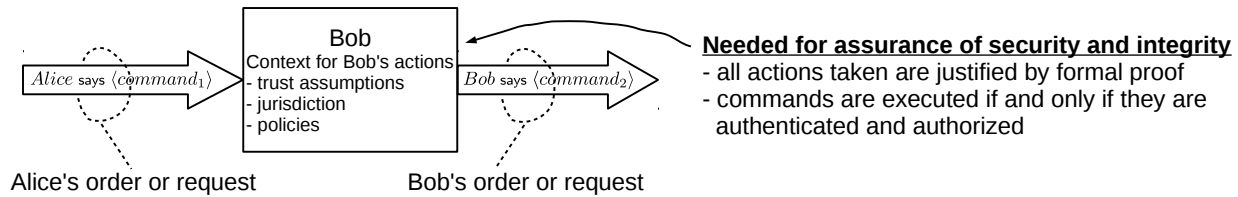
Certified Security by Design (CSBD) is an approach intended to support security by design from the start and to provide credible evidence that security claims are true. The goals of CSBD are:

1. **Complete mediation**—authenticating and authorizing—all commands at all levels from high-level concepts of operations down to transition systems realized as state machines in hardware, and
2. **Formal proofs of integrity and security that are easily and rapidly verified by third parties**, using computer-assisted reasoning tools. This enables system definitions, theorems, and proofs to be reproduced, verified, and reused by people other than the designers. Rapid verification, reproduction, and reuse inspires confidence that the system is secure and has integrity.

### 1.4.1 Concepts of Operation

CSBD supports the use of *concepts of operation* or CONOPS as a means of describing how a system accomplishes its purpose or mission. There are at least two relevant definitions of CONOPS.

1. The Institute of Electrical and Electronics Engineers (IEEE) Standard 1362 [IEE, 1998] defines a CONOPS as an expression of the “characteristics for a proposed system from a user’s perspective. A CONOPS also describes the user organization, mission, and objectives from an integrated systems point of view.”
2. The US military has a similar definition of CONOPS in Joint Publication 5-0, Joint Operational Planning [JP5, 2011]. For military leaders planning a mission, a CONOPS describes “how the actions of components and organizations are integrated, synchronized, and phased to accomplish the mission.”

**Figure 1.3** Flow of Command and Control (C2) for a Simple CONOPS

Simply put, a CONOPS describes who does what, when, where, and why. A CONOPS describes the flow of command, control, communications (C3) and actions taken by system components to accomplish a mission. CONOPS are used to describe systems at various levels of detail.

1. A high-level CONOPS might define a system's behavior in terms of roles and the actions taken by each role, e.g., commanders and their orders combined with operators and their actions.
2. A mid-level CONOPS might refine a high-level CONOPS by adding people and processes authorized to act in the roles defined at the high level.
3. A low-level CONOPS might include the cryptographic keys and cryptographic functions used to authenticate people and processes acting in their assigned roles.

Figure 14.1 shows a diagram of a simple CONOPS. Here is its interpretation.

1. The flow of command and control in this figure is from left to right. Alice issues a command by some means (speaking, writing, electronically, telepathy, etc.). This is symbolized by

*Alice says <command<sub>1</sub>>.*

2. The box in the center labeled **Bob** shows Bob receiving Alice's command on the left. Inside the box are the things Bob "knows", i.e., the context within which he attempts to justify acting on Alice's command. The context might include a policy that if Bob receives a particular command, such as *go*, then he is to issue another command, such as *launch*. Typically, before Bob acts on Alice's command, his operational context includes statements or assumptions such as Alice has the authority, jurisdiction, or is to be believed on matters related to the command she has made.

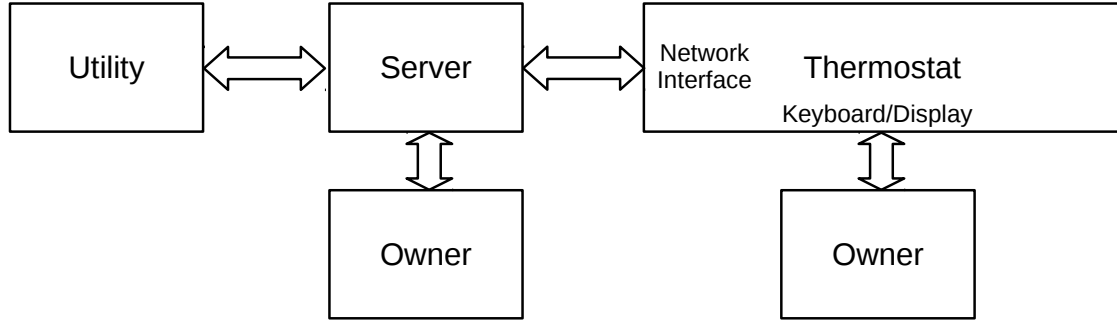
3. The arrow coming from the right hand side of the box shows Bob's statement or command, which is symbolized by

*Bob says <command<sub>2</sub>>.*

4. What Figure 14.1 shows is one C2 sequence starting from left to right. Bob gets an order from Alice. Bob decides based on Alice's order and what he knows (the statements inside the box), that it is a good idea to issue *command<sub>2</sub>*. This is symbolized by

*Bob says <command<sub>2</sub>>.*

Regarding the comment in Figure 14.1, for assurance what we want is a logical justification of the actions Bob takes given the order he receives and the context within which he is operating. For us, logical justifications are *proofs in mathematical logic*.

**Figure 1.4** A Networked Thermostat and Its Operating Environment

Security vulnerabilities often result from inconsistencies among CONOPS at various levels of abstraction. Supervisors might assume only authorized operators are able to launch an application, whereas the application itself might incorrectly trust that all orders it receives are from authorized operators and never authenticate the inputs it receives. Any *design for assurance* methodology must address authentication and authorization in order to *avoid vulnerabilities* due to unauthorized access or control. Rigorous assurance requires mathematical models and proofs. Our intent is to provide a rigorous methodology to achieve security by design.

### 1.4.2 A motivating example

Consider Figure 1.4, which shows a thermostat that is controlled by manually at its physical location, and over a network through a server. Such devices are common in the cyber-physical infrastructure that is the Internet of Things. Notice that Figure 1.4 also shows a *Utility* potentially controlling the *Thermostat* through the *Server*, as well as the *Owner*.

The benefits of a networked thermostat include:

1. The convenience afforded thermostat *Owners* to remotely control the temperature of their houses.
2. The ability of *Utilities* to take control over homeowners' thermostats while homeowners are away to reduce electricity usage (e.g., air conditioning usage during hot summer months) in order to reduce peak demand. This avoids having to bring on-line costly and polluting power plants.

Thinking through some of the details, we have three use-cases.

1. The *Owner* issues commands via the thermostat's keyboard.
2. The *Owner* issues commands to the thermostat via the *Owner's* account on the *Server*.
3. The *Utility* issues commands to the thermostat via the *Server*.

At a high level, the thermostat commands are as follows.

1. **Setting** the temperature *value*. This command has security considerations as losing control over the temperature potentially is a threat to the safety of lives and property.
2. **Enabling** the *Utility* to exercise control over setting the temperature. This command has security considerations as *Owners* want to make sure they have the ultimate authority over their thermostat.



3. **Disabling** the *Utility* to exercise control over setting the temperature. This command has similar security considerations as the command used to enable the *Utility* to alter the thermostat's temperature setting.
4. Reporting the **Status** of the thermostat, which is displayed on the thermostat and sent to the *Server*. This command does not alter the thermostat's temperature setting or operating mode.

### Thermostat Security CONOPS

At this point in conceptualizing the networked thermostat, we need to consider the concepts we use to secure integrity of the thermostat's operations. We incorporate the following concepts into our design:

- Authenticating principals issuing commands using mechanisms such as (1) userids and passwords associated with *Owner* accounts on the *Server*, and (2) cryptographically signed messages from the *Server* to the thermostat and from the *Utility* to the *Server*,
- Authorizing principals issuing commands by making explicit the context in which authorization is done, i.e., public-key certificates, root trust assumptions on keys and jurisdiction, and policies stating what actions are taken in particular circumstances, and
- Executing or trapping commands based on a principal's authority and the security sensitivity of the command they are attempting to execute.

### Assurance Requirements

The description of the networked thermostat example and the goals of CSBD lead us to the following requirements to realize the goals of CSBD.

- A C2 calculus used to reason about access-control decisions. The calculus we use is fully described in [Chin and Older, 2010] and is an extension and modification of an access-control logic for distributed systems [Abadi et al., 1993].
- Computer-assisted reasoning tools to (1) formally verify all proofs and assurance claims, and (2) enable rapid reproduction of all results by third parties and certifiers. We use the Cambridge University HOL-4 (Higher Order Logic) theorem prover [Gordon and Melham, 1993]. It is freely available and has been in use since 1987.
- A model of idealized cryptographic operations and their properties implemented in HOL.
- Models of state machine transition systems incorporating authentication, authorization, next-state functions, and output functions as parameters in support of security and to avoid state explosion. Our networked thermostat illustration builds upon the foundations of virtual machines, in particular [Popek and Goldberg, 1974].

#### 1.4.3 What you will learn

Motivated by the networked thermostat in Figure 1.4 and the assurance requirements above, our lessons and laboratory exercises are organized as follows.

**Part II:** Introduction to ML. Labs in this part teach you how to program in the functional programming language ML. This is the underlying language of the HOL theorem prover.

**Part III:** Introduction to HOL. Labs in this part teach you the basics of using the HOL theorem prover. This includes both forward proofs using inference rules and goal-oriented proofs using tactics.

**Part IV:** Access-Control Logic in HOL. Labs in this part teach you to use the inference rules and tactics that are written for the access-control logic embedded within HOL. This allows you to translate your access-control logic proofs found in the textbook *Access Control, Security, and Trust: A Logical Approach* [Chin and Older, 2010], into HOL theories, theorems, and proofs.

**Part V:** Cryptographic Components. Labs in this part teach you how to use an algebraic theory of cryptographic components in HOL. These components include: symmetric and asymmetric keys, encryption, decryption, hashes, and digital signatures.

**Part VI:** Transition Systems. Labs in this part introduce a parametric approach to describing *secure* state machines (infinite and finite) in HOL. We show how to incorporate security conditions into the transition relations describing state machines. This allows us to prove that transitions occur if and only if security conditions, i.e., authentication and authorization, are satisfied.

The above parts provide the infrastructure needed to do certified security by design (CSBD) in terms of concepts, tools, and techniques. Throughout the labs, we will introduce, define, and refine, the networked thermostat. Nevertheless, the most crucial part of the CSBD system is you, the actual user. Tools and theories by themselves are not useful. Without users and designers, nothing happens.

Let's get to work!

## **Part II**

# **Lab Exercises: Introduction to ML**



# Getting Started with Linux, ML, and Emacs

---

This lab introduces you to some of the key tools we use. Our goal is to *use formal methods (logic, languages, and tools) to assure the security and correctness of the systems we build*. The objective is to equip you with the capability of using formal methods to specify, design, and verify secure systems. We approach this objective with the following priorities:

1. *Learn by doing.* You must be able to use mathematics and logic to specify, design, and verify your systems *yourself*.
2. *Build upon the work of others.* The advantages of using theorem provers in general and HOL in particular are the availability of theory libraries upon which you can build your designs and theories with assurance. There is no need to re-invent basic theories.
3. *Enable others to reproduce and check your results quickly and easily.* Others can build upon your work with assurance if your development process uses computer-aided design and verification tools. This is similar to libraries of subroutines in software and macrocells in hardware.

A word about the *Emacs* editor. The tools we use in the assurance labs are ML, HOL, and  $\text{\LaTeX}$ . Within  $\text{\LaTeX}$  you will use *beamer* for presentations as well as for generating reports. All these tools have an Emacs interface. If we were restricted to only one tool, we might be able to get by without Emacs. However, as we move back and forth among ML, HOL, and  $\text{\LaTeX}$ , using Emacs is a significant advantage as it (1) provides a consistent editing interface, and (2) can execute ML, HOL, and  $\text{\LaTeX}$  interpreters.

## 2.1 Getting set up

We assume you are running a current version of Linux.

1. To navigate and open files, click the **Files** icon that looks like an open file cabinet near the top of the applications ribbon at the left of your screen. This brings up *Nautilus*, the Linux equivalent of Windows Explorer.
2. Much like Windows and OS-X, click on the icons to execute the applications.
3. On the applications ribbon, click on the **Terminal** application.
4. You should see a terminal window. You can adjust the size of the window, the font, and other settings via the application's **Edit** item  $\rightarrow$  **Profile Preferences**, which is displayed in the top ribbon on your display.

Our first objective is to set up a subdirectory for our labs.

1. Click on the **Files** icon (or use the shortcut `Ctrl-N`).

2. Click on **Documents** under **Places**. This displays the contents of the **Documents** folder.
3. If the **COURSES** folder is already present, click on it to take you inside. If not, create the folder within **Documents** by *right clicking* your mouse and selecting **New Folder**, or going to the menu ribbon on **Files** and clicking on **Files** → **New Folder**. Create the folder **COURSES**.
4. Go into the **COURSES** folder and create the **csbd** folder.
5. Within the **csbd** folder, create a folder called **ML**.
6. Within the **ML** folder, create another folder **MLLab1**.

When we have a lot of folders, it is convenient to be able to go directly to a folder without traversing the entire subdirectory structure. We can do this using *Bookmarks*. To bookmark a folder, do the following:

1. Click on **Files** and go *into* the folder you wish to bookmark.
2. Go to the **Files** menu ribbon at the top of your display, and click on **Bookmarks** → **Bookmark this Location**.
3. Doing the above will add the folder's name in the **Bookmarks** section of **Files**. Once a bookmark is there, you can select it using your left mouse button and move it within the **Bookmarks** section. You can *rename* a bookmark by *right-clicking* on it and clicking on **Rename**.

## 2.2 Looking around using Linux commands within a terminal

Frequently, it is much more convenient to use the command-line interface provided by a *terminal* window. Both Linux and ML are case sensitive (i.e., capitalization matters), so make sure you type things exactly as stated throughout the rest of this writeup.

1. Click on the **Terminal** icon to open a terminal (or use `Ctrl+Alt+T` as a shortcut)
2. Type to the terminal prompt:

```
ls -F ↵
```

(The “↵” means “hit return.”) You’ll see the names of files and subdirectories. These are files and directories in your home directory (a.k.a, home folder). The command “ls” stands for *list directory*, and the “-F” is a specific option for how the directory listing should be formatted:

*An ending “/” indicates a directory and an ending “\*” indicates an executable file; an ending “~” indicates a link to another file. The “/” and “\*” and “~” are decorations and not really part of a name.*

3. Now type to the terminal prompt:

```
ls ↵
```

You should see the same names as before, but without the trailing “/” and “\*” characters that the “-F” provided previously.

4. To change directories, use `cd`

```
cd Documents/COURSES/csbd/ML/MLLab1 ↵
```

puts you into the MLLab1 subdirectory, starting from your home directory.

5. The “pwd” command (for *print working directory*) tells you the name of the current directory you are in.
6. For times you need superuser (a.k.a. administrator) privileges, use `sudo <command>`, or `sudo -i`. You will be asked to type in your password to authenticate your privileges.

If you execute `sudo -i`, you will become the **root** superuser and be placed in the root subdirectory **/root** as shown below.

```
skchin@skVM:~/Documents/COURSES/csbd$ sudo -i
[sudo] password for skchin:
root@skVM:~# pwd
/root
```

1

It is **dangerous** to be the root superuser as any command will be executed, including deleting all your files. If you create a file while you are root, then only root can change it.

7. To exit superuser status, type `exit` or `CTRL-D` in the terminal.
8. To see if you are root or a normal user, type `whoami` in the terminal.

```
skchin@skVM:~/Documents/COURSES/csbd$ sudo -i
[sudo] password for skchin:
root@skVM:~# whoami
root
root@skVM:~# exit
logout
skchin@skVM:~/Documents/COURSES/csbd$ whoami
skchin
skchin@skVM:~/Documents/COURSES/csbd$
```

2

**Unless otherwise directed, everything we do in our labs is done in user mode. If you have entered superuser mode, make sure you exit it now and are just a regular user.**

## 2.3 Starting HOL and Evaluating ML Expressions

1. For something a bit more interesting, type to the terminal prompt:

```
hol ↵
```

You will see something like:

---

```
skchin@skVM: ~/Documents/COURSES/csbdfall2015/book/ML$ hol
-----
HOL-4 [Kananaskis 10 (stdknl, built Wed May 20 23:02:10 2015)]

For introductory HOL help, type: help "hol";
To exit type <Control>-D (*not* quit());
-----

[loading theories and proof tools ..... ]
[closing file "/usr/local/share/HOL/tools/end-init-boss.sml"]
-
```

---

2. To the “-” prompt, type:

```
1+1; ↵
```

and you should see 2 as the answer.

3. Now type to the “-” prompt (note CTRL-D means hold the `ctrl`-key down and then press the `D`-key):

```
:help "hol" ↵
:CTRL-D
```

The “`help "hol"`” command gives you the various theories and libraries built into HOL, and the “CTRL-D” quits HOL.

## 2.4 Starting Emacs

We use the Emacs editor. One of the advantages of using the Emacs editor is its capability to execute applications and processes within an editing environment. A single Emacs session can run multiple windows; you can cut and paste among them. Emacs has special interfaces (in the form of keyboard shortcuts and menu buttons) for the HOL theorem prover, the  $\text{\LaTeX}$  scientific type setting system we use, and programming languages such as Haskell.

Next, we provide you with a gentle introduction to Emacs. Its capabilities developed over more than three decades are voluminous. For more information on Emacs, you can consult the various on-line references that are freely available.

1. Make sure you are **not** in superuser (root) mode and that you are in the subdirectory for your ML lab exercises, such as `~/Documents/COURSES/csbdfall2015/book/ML/MLLab1`
2. Type to the terminal prompt:

```
touch simple.sml ↵
emacs simple.sml ↵
```

What `touch` does is update the timestamp associated with the access and modification times of the file appearing in the argument (in this case `simple.sml`) to the current time. If the file does not exist, an empty file is created. For more information on this command (and any Linux command) you can type `man touch` ↵ in a terminal window.

What we have done is start up the Emacs text editor on a new file named `simple.sml`. The “`.sml`” is the extension for ML files (the “`s`” in “`.sml`” stands for “standard”).



You may see some text (“Welcome to GNU Emacs...”) in the bottom half of the window. If so, click on the line “Dismiss this startup screen” and “never show this message again”; you should end up with a single empty buffer.

3. Click on the Emacs window and type into that window:

```
fun plus x y = x+y; ↵
↵
fun times x y = x*y; ↵
```

The arrow keys move you around the text, and `delete` does its usual thing.

What you have done is defined two functions in the ML programming language. The function `plus x y` takes two integers `x` and `y` and adds them. The function `times x y` takes two integers `x` and `y` and multiplies them.

Next, we want to save our work.

4. 🖱️ **Read this part *completely* before trying it out!** Type to Emacs

C-x C-s

“C-x” means hold the control-key and the x-key at the same time. “C-s” means hold the control-key and the s-key at the same time. This saves the file.

[Alternatively, go to the **File** menu, and then select **Save**.] Notice, that to the right of **Save** in the menu is the corresponding keyboard short-cut C-x C-s. This is one way to learn and remember the short-cuts.

5. Now, we start a HOL session within our Emacs frame housing `simple.sml`. Locate the **HOL** button on the menu ribbon at the top of your Emacs frame, and do the following sequences of clicks.

**HOL** **Process** **Start HOL-vertical split**

Note that the associated keyboard short-cut is M-h 2. This means press and release the `esc` key followed by pressing and releasing the `h` and `2` keys—unlike C-x, where you hold both the control-key and the x-key at the same time.

The Emacs window should split in half. What you will see is something like:

```
-----
HOL-4 [Kananaskis 10 (stdknl, built Wed May 20 23:02:10 2015)]

For introductory HOL help, type: help "hol";
To exit type <Control>-D (*not* quit();)
-----

[loading theories and proof tools ..... ]
[closing file "/usr/local/share/HOL/tools/end-init-boss.sml"]
-
```

6. To send what you have typed in `simple.sml` to the HOL interpreter, within the `simple.sml` window, press and hold your left mouse button and highlight the region:

```
fun plus x y = x + y;
fun times x y = x*y;
```


When the region is highlighted, release the mouse button and do the following sequence of clicks. Notice that the corresponding keyboard short-cut is [M-h M-r](#).

**HOL** **Process** **Send region to HOL**

What you should see is the following:

```
- > val plus = fn : int -> int -> int
- > val times = fn : int -> int -> int
- -
```

7. Within the `*HOL*` window right after the “-” prompt, you can type

`plus 1 2;` 

What you should see is:

```
- - plus 1 2;
> val it = 3 : int
```


Alternatively, we could have typed `plus 1 2;` within the file `simple.sml`, highlighted it with our left mouse button, and sent the region to HOL by doing **HOL** **Process** **Send region to HOL** or [M-h M-r](#).

At this point, we have defined two functions and tested one of them.

8. When debugging programs, it is helpful to have line numbers. To turn on (and off) line numbers in a buffer do [M-x linum-mode](#), where [M-x](#) means press *and release* the ESC-key and then type x. For toggling line numbers on and off we do:

(a) Press and release ESC

(b) Press and release x

(c) Type `linum-mode` 

9. Now type to Emacs: C-x C-c and answer **yes** to the question it asks you. The Emacs window should go away.

## 2.5 Exercises

In the previous parts, we walked you through various tasks. Here you get to carry out a similar set of tasks on your own. If you can’t remember a command, there is an index of what we covered at the end of this lab.

### Exercise 2.5.1

1. Start up Emacs with a fresh file `ex-2-5-1.sml`.

- 
2. In Emacs, insert the following text into `ex-2-5-1.sml`, where `(*` and `*)` are used to surround comments in ML.

```
(* Name:   fill in your name *)  
(* Email: fill in your email address *)  
  
fun timesPlus x y = (x*y, x+y);
```

3. Start HOL inside of Emacs, highlight the definition of `timesPlus`, and send the region to HOL.
4. Evaluate the expression `timesPlus 100 27` within HOL. If you've done things correctly, you should get a pair of integers as a result. Note: when you start HOL within Emacs, a second window opens below or on the right of your source code. This is the `*HOL*` buffer. Move your cursor to this buffer by using your mouse or by typing C-X o, which moves the cursor among the various Emacs buffers/windows.
5. Kill the HOL process while preserving the `*HOL*` window by moving your cursor to the `*HOL*` window and typing C-D. Save the contents of the `*HOL*` window under the name `ex-2-5-1.trans`.
-

## ***What was that command?***

### **Linux Summary**

<b><i>ls</i></b>	lists the files in a directory
<b><i>cd</i></b>	changes directory
<b><i>mkdir</i></b>	makes a new directory
<b><i>pwd</i></b>	prints the name of the current directory
<b><i>cp</i></b>	copies files and directories
<b><i>mv</i></b>	moves or renames a file
<b><i>rm</i></b>	deletes a file
<b><i>sudo</i></b>	executes a command as superuser
<b><i>whoami</i></b>	gives the effective userid (particularly useful if you've done sudo)
<b><i>pwd</i></b>	gives the name of the current working directory
<b><i>hol</i></b>	starts up HOL
<b><i>emacs</i></b>	starts up emacs
<b><i>man</i></b>	an interface to the on-line reference manuals
<b><i>firefox</i></b>	starts up firefox

### **Emacs Summary**

<u>C-x C-s</u>	saves a buffer as a file
<u>C-x C-c</u>	quits emacs
<u>C-x C-f</u>	loads a file into Emacs
<u>C-x o</u>	moves the cursor to another Emacs window/buffer
<u>M-x linum-mode</u>	toggles line numbers on and off in a buffer
<u>C-g</u>	quits a command
<u>C-x u</u>	undoes the last operation
<u>M-w</u>	copies highlighted region
<u>C-w</u>	cuts highlighted text
<u>C-y</u>	pastes (yanks) text most recently copied or cut to where the cursor is

# An Introduction to ML within HOL

---

In this lab, we learn the basics of the ML functional programming language. The name ML stands for *Meta Language*. While ML is a functional programming language in its own right, ML's original purpose is to manipulate and reason about theorems in *higher-order logic*. We will learn a lot more about this in the coming labs. For now, our introduction to ML focuses on defining expressions in the form of values, functions, their types, and commonly used data types, such as pairs, tuples, and lists. As ML is the basis for using HOL, it is crucial that you are proficient in the basics of ML.

In the work we do, we use ML in the context of HOL. What this means is that we will use the HOL interpreter (usually invoked within Emacs) instead of the Moscow ML interpreter, `mosml`. We need a fairly small subset of ML to use HOL. Nevertheless, it is useful to have easily accessible references on ML to find answers to your questions. Some suggested references (and how to find them) are listed below.

### Where to find more information on ML and HOL

1. *Poly/ML Documentation*—the Poly/ML distribution site is <http://www.polyml.org/>. You will find FAQs, documentation, and downloads.
2. *Programming in Standard ML*, Robert Harper, Carnegie Mellon University—this freely available reference licensed under the Creative Commons <http://creativecommons.org/licenses/by-nc-nd/3.0/us/> is a draft textbook on ML with numerous examples. The textbook is available at <http://www.cs.cmu.edu/~rwh/isml/book.pdf>.
3. *HOL Interactive Theorem Prover*—HOL is freely available and is thoroughly documented with tutorials, descriptions, examples, and reference manuals. Your VM is equipped with local documentation. The HOL distribution site is <https://hol-theorem-prover.org/>.
4. *ML for the Working Programmer*, Lawrence Paulson, Cambridge University Press, 1996 [Paulson, 1996]. This book is not online but is an excellent textbook.

We use functions written in ML to define new logical datatypes, definitions, theorems, proof goals, and proofs in HOL. This means we need to be able to:

1. apply existing ML functions to arguments to compute values, manipulate the components of existing data structures, or return some component of a data structure,
2. define new functions in ML, and
3. define new datatypes in ML.

In the sections that follow, we focus on the basics of values, types, functions, and datatypes in ML.

### 3.1 Values and Types in ML

**Starting HOL within Emacs:** We begin by setting up an empty file named *mlLab2.sml*, opening it within Emacs, and starting HOL within Emacs. Do the following:

1. Create the empty file *mlLab2.sml* using `touch` or right-click your mouse and select **New document** within your personal subdirectory, e.g., `~/Documents/COURSES/csbd/ML`, rename the file to *mlLab2.sml*.
2. Click on the **Files** icon in your Linux menu ribbon and navigate to the subdirectory with you have created *mlLab2.sml*.
3. Open *mlLab2.sml* in Emacs. **Right** click on *mlLab2.sml* and click on either **Open with GNU Emacs**, if Emacs is your default editor, or click on **Open With** followed by **Emacs**.
4. Once the Emacs frame is visible, you will note that in the top menu bar is the **HOL** button. Click on it, click on **Process**, and click on **Start HOL – vertical split**. This will do a vertical split of your Emacs frame. What appears in the lower half of Emacs is the *\*HOL\** window. Its contents is shown in the numbered session box below.

1

```

-----
HOL-4 [Kananaskis 10 (stdknl, built Wed May 20 23:02:10 2015)]


For introductory HOL help, type: help "hol";
To exit type <Control>-D (*not* quit());
-----

[loading theories and proof tools ..... ]
[closing file "/usr/local/share/HOL/tools/end-init-boss.sml"]
-

```

**Basic values and their types:** We will use the file *mlLab2.sml* to keep a record of what we do in ML. By so doing, you can return later to the file, run HOL, reproduce everything, modify, and add to it. This kind of experimentation helps you acquire the capability to use ML.

Do the following:

1. With your left mouse button, click the top left hand corner of the white space in the *mlLab2.sml* window. This moves the cursor into the top left hand corner of *mlLab2.sml*.
2. Enter what is shown below in the shadow box. Remember from the previous chapter that comments are surrounded by `( * and * )`, and `;`  sends the expression to the left of the semicolon to the ML interpreter for evaluation.

```

(*****)
(* Basic values and types *)
(*****)
1;
true;
"This is a string";

```

3. Enter `C-x C-s` to save the file, or click **File** **Save**.

4. Highlight **everything** you typed in *mlLab2.sml* with your left mouse button.
5. Enter **M-h M-r** or click **HOL** **Process** **Send region to HOL**. What you should see as a result in your *\*HOL\** window is shown in the numbered session box below.

```
- > val it = 1 : int
- > val it = true : bool
- > val it = "This is a string" : string
- -
```

2

Notice that none of the comments appeared in the *\*HOL\** window. The top three lines correspond to evaluating the three ML expressions:

1. 1
2. true
3. "This is a string"

The meaning of lines in the numbered session box above is as follows:

1. `val it = 1 : int`—“it” refers to the expression “1” in *mlLab2.sml*. Its value is an *integer of value 1*.
2. `val it = true : bool`—“it” refers to the expression “true” in *mlLab2.sml*. Its value is a *boolean of value true*.
3. `val it = "This is a string" : string`—“it” refers to the expression “"This is a string"” in *mlLab2.sml*. Its value is a *string of value "This is a string"*.

You can interact directly with the interpreter expression by expression by either typing expressions directly into the *\*HOL\** window at the last “-” prompt at the bottom, or by copying and pasting expressions from your sml source file. Usually, when developing ML programs and HOL theories, we will copy expressions from our sml files to our *\*HOL\** window. Do the following:

1. Highlight `1;` in *mlLab2.sml* using the left mouse button.
2. When highlighted, release the left mouse button and enter **M-w**, or click **Edit** **Copy**. This copies `1;`.
3. Go to the bottom of your *\*HOL\** window by using the scroll bar on the right hand side of the *\*HOL\** window, using the arrow keys, or entering **M->**, i.e. sequentially enter the `esc` followed by `>` (the shift “.”) key.
4. Paste what you have copied after the last ML prompt “-” by entering **C-y** or clicking **Edit** **Paste**.
5. Enter **↵** to send the expression to the interpreter.
6. Copy, paste, and enter the remaining expressions `true;` and `"This is a string";` from *mlLab2.sml* individually into *\*HOL\**.
7. What you should see is shown in the numbered session box below.

```

- - 1;
> val it = 1 : int
- true;
> val it = true : bool
- "This is a string";
> val it = "This is a string" : string

```

3

The major difference with this input approach versus highlighting the entire region as we did at first, is the appearance of the ML source input expression. The corresponding values remain unchanged.

When developing ML programs or HOL proofs, we will often send entire regions of source code to HOL when we have already “debugged” them. When adding or modifying new code, we often interpret each new expression individually, as we have just shown above, to make sure all is as expected or to find errors.

**Everything has a type in ML:** ML is a *strongly typed* language. This means that everything such as basic values such as integers and booleans, to functions have a *type*. Types are like *shapes*. Just like the saying “you can’t put a square peg in a round hole,” types help us avoid making errors, such as trying to add non-numbers to numbers—something that is nonsensical.

Often, when developing ML programs or HOL proofs, we will explicitly include the type of expressions for precision and to check our thinking. When we do this, we say we are including the *type signature* of the expression. Expression with their type signatures have the form `exp : ty`, where `exp` is the expression, and `:` is the punctuation separating the expression from its type `ty`.

As an illustration of explicit typing, do the following:


1. Add to *mlLab2.sml* what is shown below in the shadow box.

```

(*****
(* Explicitly specifying types in expressions *)
*****)
2 : int;
false : bool;
"Another string" : string;

```

Make sure you save the additions by doing C-x C-s.

2. Copy (M-w) each expression individually and paste (C-y) it into the HOL interpreter. Don’t forget to hit  after each expression and its type signature.
3. What you should see is in the numbered session box below.

```

- 2 : int;
> val it = 2 : int
- false : bool;
> val it = false : bool
- "Another string" : string;
> val it = "Another string" : string

```

4

**Basic type errors:** Many times when we are writing complicated expressions, we are tired, or just plain wrong, we make type errors, i.e., we think an expression has one type when it is really another. What happens when we have type errors? Basically, the interpreter tells us we are wrong and does its best to point out where the error is.

To illustrate what happens when we make type errors, do the following:



1. Add to *mLab2.sml* what is shown below in the shadow box.

```
(*****)
(* Basic type errors *)
(*****)
0 : bool;
false : string;
"1" : int;
```

The above code mistakenly gives the integer 0 the type `bool`, the boolean `false` the type `string`, and the string `"1"` the type `int`.

2. Copy each individual expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- 0 : bool;
! Toplevel input:
! 0 : bool;
! ^
! Type clash: expression of type
!   int
! cannot have type
!   bool
- false : string;
! Toplevel input:
! false : string;
! ^^^^^
! Type clash: expression of type
!   bool
! cannot have type
!   string
- "1" : int;
! Toplevel input:
! "1" : int;
! ^^^
! Type clash: expression of type
!   string
! cannot have type
!   int
```

5

Each of the three expressions has the same kind of typing error, known as a type clash.

1. ML points to 0 and says it has type `int` and cannot be type `bool` as written in `0 : bool`.
2. ML points to `false` and says it has type `bool` and cannot be type `string` as written in `false : string`.
3. ML points to `"1"` and says it has type `string` and cannot be type `int` as written in `"1" : int`.

Newcomers to strongly typed languages such as ML are sometimes frustrated by the type checker. One thing to remember about HOL:

**HOL is never wrong!**

If HOL complains, then there is an error or an ambiguity that prevents it from deriving a meaningful value or interpretation of an expression. While initially frustrating, detecting errors early in development is much less costly in terms of time and risk than correcting errors later, e.g., after a system is deployed.

## 3.2 Compound Values

Thus far we have only considered simple values, such as integers, booleans, and strings. In this section, we learn about *pairs*, *tuples*, and *lists* in ML.

**Pairs and tuples:** *Pairs* and *n-tuples* in ML are constructed using the comma “,”. Pairs have two components,  $(x,y)$  where  $x$  is the first element of the pair and  $y$  is the second element of the pair. Pairs are two-element  $n$ -tuples.  $N$ -tuples are any size where  $N$  is any number. For example,  $(\text{"B"}, (1, 2), \text{false}, 12)$  is a 4-tuple whose elements in order are:

1. the string "B",
2. the pair of integers  $(1, 2)$ ,
3. the boolean `false`, and
4. the integer 12.

To see the interpretation of  $n$ -tuples in ML, do the following:

1. Add to *mlLab2.sml* what is shown below in the shadow box.

```
(*****)
(* a pair consisting of a string and an integer *)
(*****)
("A", 0);

(*****)
(* a 3-tuple consisting of a string, integer, and boolean *)
(*****)
("B", 1, false);

(*****)
(* a 4-tuple *)
(*****)
("B", (1, 2), false, 12);
```

2. Copy each individual expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- ("A", 0);
> val it = ("A", 0) : string * int
- ("B", 1, false);
> val it = ("B", 1, false) : string * int * bool
- ("B", (1, 2), false, 12);
> val it = ("B", (1, 2), false, 12) : string * (int * int) * bool * int
```

6

The interpretation and values of the expressions shown in the numbered session box above is as follows.

1. The value of  $(\text{"A"}, 0)$  is  $(\text{"A"}, 0)$ , whose type corresponds to a Cartesian product ( $\text{string} \times \text{integer}$ ), represented by `string * int` in ML.
2. The value of  $(\text{"B"}, 1, \text{false})$  is  $(\text{"B"}, 1, \text{false})$ , whose type is a 3-tuple ( $\text{string} \times \text{integer} \times \text{boolean}$ ) represented by `string * int * bool` in ML.

3. The value of `("B", (1, 2), false, 12)` is `("B", (1, 2), false, 12)`, whose type is a 4-tuple  $(string \times (integer \times integer) \times boolean \times integer)$  represented by `string * (int * int) * bool * int` in ML.

To illustrate what happens when there are type errors within an n-tuple, do the following:

1. Add to `mlLab2.sml` what is shown below in the shadow box.

```
(*****)
(* A type error within an n-tuple *)
(*****)
(1, false, "A") : bool * string * int;
```

2. Copy the expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- (1, false, "A") : bool * string * int;
! Toplevel input:
! (1, false, "A") : bool * string * int;
! ~~~~~
! Type clash: expression of type
!   int * bool * string
! cannot have type
!   bool * string * int
```

7

Similar to the previous examples with simple values and type clashes, the ML interpreter reports a type clash. Specifically, the expression we said was of type  $boolean \times string \times integer$ , the interpreter deduces that it is really of type  $integer \times boolean \times string$ .

**Lists:** Lists are another built-in ML data type. The facts to remember about lists are:

1. All list elements must be the same type, and
2. There are only two kinds of lists: (a) the empty list `[]`, and (b) non-empty lists whose structure is `head :: tail`, where `head` is an element of type  $\alpha$ , and `tail` is a list of elements of type  $\alpha$ , i.e., an  $\alpha$  list.

To see lists in ML, do the following:

1. Add to `mlLab2.sml` what is shown below in the shadow box.

```
(*****)
(* A list of integers in ML *)
(*****)
[1, 2, 3, 4, 5];

(*****)
(* A list of bool * int pairs in ML *)
(*****)
[(false, 0), (false, 4), (true, 1), (false, 3)];

(*****)
(* An ill-formed list where the elements differ in type *)
(*****)
[1, true];
```

2. Copy each ML expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

8

```

- [1,2,3,4,5];
> val it = [1, 2, 3, 4, 5] : int list
- [(false,0), (false, 4), (true, 1), (false, 3)];
> val it = [(false, 0), (false, 4), (true, 1), (false, 3)] : (bool * int) list
- [1,true];
! Toplevel input:
! [1,true];
!   ^^^^
! Type clash: expression of type
!   bool
! cannot have type
!   int

```

ML's interpreter returns the value and type of the first two lists as expected: the first list is a list of integers `int list`, and the second list is a list of `bool * int` pairs, i.e., `(bool * int) list`. The third list is ill-formed in that the first element of the list is the integer 1 and the second element is a boolean. As lists are composed of elements of the same type, the ML interpreter complains that a `bool` cannot be an element of an `int list`.

### 3.3 Value Declarations

Value declarations name values, functions, and types, among other things. In this chapter, we will show how to name *constants*. In the next chapter, we will show how to name functions.

**Simple value declarations:** The general pattern for declaring constants is: `val name = expression`. To see how to name constants in ML, do the following:

1. Add to *mLlab2.sml* what is shown below in the shadow box. **Note: in the definition of `x2`, the negative sign is “tilde”  $\sim$ , i.e.,  $\sim 42$ . If you use a regular minus sign “-”, you will get an error because it expects formulas of the form  $x - y$ .**

```

(*****)
(* Giving names to constants *)
(*****)
val x1 = 42;
val x2 = ~42;
val x3 = x1 + x2;
x1;
x2;
x3;
val y1 = (true,12);
val y2 = (false, 0);
val y3 = [y1,y2];
y1;
y2;
y3;

```

2. Copy each ML expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

9

```

- val x1 = 42;
> val x1 = 42 : int
- val x2 = ~42;
> val x2 = ~42 : int
- val x3 = x1 + x2;
> val x3 = 0 : int
- x1;
> val it = 42 : int
- x2;
> val it = ~42 : int
- x3;
> val it = 0 : int
- val y1 = (true,12);
> val y1 = (true, 12) : bool * int
- val y2 = (false, 0);
> val y2 = (false, 0) : bool * int
- val y3 = [y1,y2];
> val y3 = [(true, 12), (false, 0)] : (bool * int) list
- y1;
> val it = (true, 12) : bool * int
- y2;
> val it = (false, 0) : bool * int
- y3;
> val it = [(true, 12), (false, 0)] : (bool * int) list

```

The above session show that constant names, such as `x1`, `x2`, `x3`, `y1`, `y2` and `y3` can be assigned simple values such as `int`, and compound values such as `bool * int` and `(bool * int) list`. Once a name is assigned a value in a HOL session, that value remains associated with the name until it is changed. Note: later on we will show the use of local assignment using `let` statements.

If a constant name that is undefined appears in an expression, HOL will complain. To see this, do the following:

1. Add to *mLlab2.sml* what is shown below in the shadow box.

```

(*****
(* If a variable name is undefined, HOL/ML complains *)
*****)
foo;

```

2. Copy the expression `foo`; into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

10

```

- foo;
! Toplevel input:
! foo;
! ^^^
! Unbound value identifier: foo

```

The ML interpreter within HOL has underlined `foo` as being undefined, i.e., unbound to any value.

Detecting type errors is unchanged with the use of names bound to values. To see this in ML, do the following:

1. Add to *mLlab2.sml* what is shown in the shadow box.

```

(*****
(* Type clashes are detected as before *)
*****)
[x1,x2,y1];

```

2. Copy the expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- [x1,x2,y1];
! Toplevel input:
! [x1,x2,y1];
!   ^
! Type clash: expression of type
!   bool * int
! cannot have type
!   int
```

11

In this case, the HOL interpreter singles out `y1` has type `bool * int`, whereas the other elements of the list are of type `int`.

**Value declarations using pattern matching:** One of the powerful benefits of declarative programming languages such as ML is the use of *pattern matching* in value declarations. Taking advantage of pattern matching produces clear succinct code.

**Pattern matching on tuples:** For example, suppose we have the 3-tuple `(1,true,"Alice")`, and we wish to assign the values `1`, `true`, and `"Alice"` to the constants `a1`, `a2`, and `a3`. Instead of devising an accessor function to return the first, second, and third elements of a 3-tuple, and apply each to assign values to `a1`, `a2`, and `a3` (which would take three separate expressions), we can do it all once with pattern matching. Specifically, we can use the following expression that takes advantage of matching the pattern of the 3-tuple.

$$(a1,a2,a3) = (1,true,"Alice").$$

This results in the assignment of `1`, `true`, and `"Alice"` to the constants `a1`, `a2`, and `a3`.

To see this in ML, do the following:

1. Add to `mLlab2.sml` what is shown in the shadow box.

```
(*****
(* Value declarations using pattern-matching *)
*****)

(*****
(* Pattern matching on a 3-tuple *)
*****)
val (a1,a2,a3) = (1,true,"Alice");
```

2. Copy the expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- val (a1,a2,a3) = (1,true,"Alice");
> val a1 = 1 : int
  val a2 = true : bool
  val a3 = "Alice" : string
```

12

The session above shows in the first line the value declaration matching the pattern of the 3-tuple. The subsequent three lines show the assignment of the 3-tuple components to the names `a1`, `a2`, and `a3`.

**Pattern matching on lists:** Pattern matching on lists works much like it does on tuples. For example, if we know that a list has exactly 4 elements, e.g.,  $[2, 4, 6, 8]$ , then we can match the pattern in the value declaration as follows to assign each of the four elements to the constants  $c1$ ,  $c2$ ,  $c3$ , and  $c4$ .

$$[c1, c2, c3, c4] = [2, 4, 6, 8].$$

To see how this works in ML, do the following:

1. Add to *mLab2.sml* what is shown in the shadow box.

```
(*****)
(* Pattern matching on lists *)
(*****)
val [c1, c2, c3, c4] = [2, 4, 6, 8];
```

2. Copy the expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- val [c1, c2, c3, c4] = [2, 4, 6, 8];
> val c1 = 2 : int
    val c2 = 4 : int
    val c3 = 6 : int
    val c4 = 8 : int
```

13

The above technique works when you know exactly how many elements are in a list. However, lists are of arbitrary size. Nevertheless, all lists have one of two forms: (1) the empty list  $[]$ , and (2) the non-empty list  $x :: xs$ , where  $x$  is a list element (e.g., an `int`),  $xs$  is a list (possibly empty) of elements of the same type as  $x$ , and  $::$  is the infix list constructor (pronounced “cons”).

In fact,  $[1, 2]$  is shorthand for  $1 :: (2 :: [])$ . To see this and other examples of the use of  $::$ , do the following:

1. Add to *mLab2.sml* what is shown in the shadow box.

```
2 :: [];
2 :: [4, 6, 8];
2 :: (4 :: (6 :: (8 :: [])));
```

2. Copy each expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- 2 :: [];
> val it = [2] : int list
- 2 :: [4, 6, 8];
> val it = [2, 4, 6, 8] : int list
- 2 :: (4 :: (6 :: (8 :: [])));
> val it = [2, 4, 6, 8] : int list
```

14

For **non-empty** lists of arbitrary size, we use pattern matching on  $::$  to assign values to the head and tail of a list. For a non-empty list, the expression is:

$$(head :: tail) = list.$$

To see how this works in ML, do the following:

1. Add the *mlLab2.sml* what is shown in the shadow box.

```
val (c :: cs) = [2,4,6,8];
val (d :: ds) = [3];
```

2. Copy each expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- val (c :: cs) = [2,4,6,8];
> val c = 2 : int
    val cs = [4, 6, 8] : int list
- val (d :: ds) = [3];
> val d = 3 : int
    val ds = [] : int list
```

15

In both expressions, the result is the head of the list (*c* and *d*) is bound to the first element of the list (2 and 3), and the remaining list tail (*[4, 6, 8]* and *[]*) is bound to *cs* and *ds*.

**Mismatches in pattern matching:** When we make mistakes in pattern matching, often it is due to mismatches in what we think the pattern is versus the actual pattern of the value given. For example, consider the two mismatches shown below. Do the following:

1. Add to *mlLab2.sml* what is shown in the shadow box.

```
(*****)
(* Mismatches in patterns *)
(*****)
val (e :: es) = [];
val (f1,f2) = (1,2,3);
```

2. Copy each expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- val (e :: es) = [];
! Uncaught exception:
! Bind
- val (f1,f2) = (1,2,3);
! Toplevel input:
! val (f1,f2) = (1,2,3);
!           ^^^^^
! Type clash: expression of type
!   'a * 'b * 'c
! cannot have type
!   'd * 'e
! because the tuple has the wrong number of components
```

16

The first expression tries to equate the pattern of a non-empty list to an empty list. An error occurs because there is no leading element in an empty list. Therefore, no binding to *e* is possible in the list *e :: es*. In the second expression, we try to equate the pattern of a pair to a 3-tuple. HOL complains because of a type clash, i.e., *int \* int* is not the same type as *int \* int \* int*.



### 3.4 Exercises

**Exercise 3.4.1** Create a file `ex-3-4-1.sml` as your sourcefile. Define the following values in ML. Please include comments similar to those in the examples we have shown in this Chapter. Execute your final source code in the HOL interpreter and create a transcript file `ex-3-4-1.trans` by saving the *\*HOL\** window in Emacs to `ex-3-4-1.trans`.

At the top of your `ex-3-4-1.sml` file, include the following comment block:

```
(*****
(* Exercise 4.4.1
(* Author: <your name>
(* Date: <date you wrote the file>
(*****)
```

**Hint:** Comment blocks are easily inserted by **HOL** **Templates** **Comment \*** in Emacs when editing `sml` files. If you **insert**, you can toggle the overwrite mode and just type in the whitespace of the comment block. Make sure you **insert** again to turn off overwrite, otherwise you might type over your previous work. Devise ML expressions for the following values and assign them to the constant names as specified.

- Devise the list of pairs `[(0, "Alice"), (1, "Bob"), (3, "Carol"), (4, "Dan")]` and assign it the name `listA`.
- Using `listA` and pattern matching, create the following value assignments: `elB` has the value `(0, "Alice")` and `listB` has the value `[(1, "Bob"), (3, "Carol"), (4, "Dan")]`
- Using `elB`, `listB`, and pattern matching, create the following value assignments: `elC1` has the value `0`, `elC2` has the value `"Alice"`, `elC3` has the value `(1, "Bob")`, `elC4` has the value `(3, "Carol")`, and `elC5` has the value `(4, "Dan")`.

**Exercise 3.4.2** Create a file `ex-3-4-2.sml` as your sourcefile. Define the following values in ML. Please include comments similar to those in the examples we have shown in this Chapter. Execute your final source code in the HOL interpreter and create a transcript file `ex-3-4-2.trans` by saving the *\*HOL\** window in Emacs to `ex-3-4-2.trans`.

At the top of your `ex-3-4-2.sml` file, include the following comment block:

```
(*****
(* Exercise 4.4.2
(* Author: <your name>
(* Date: <date you wrote the file>
(*****)
```

**Hint:** Comment blocks are easily inserted by **HOL** **Templates** **Comment \*** in Emacs when editing `sml` files. If you **insert**, you can toggle the overwrite mode and just type in the whitespace of the comment block. Make sure you **insert** again to turn off overwrite, otherwise you might type over your previous work.

- Insert the following code into your `ex-3-4-2.sml` file:

```
val (x1,x2,x3) = (1,true,"Alice");
val pair1 = (x1,x3);
val list1 = [0,x1,2];
val list2 = [x2,x1];
val list3 = (1 :: [x3]);
```

- Evaluate each of the assignments in the order in which they appear in HOL. Store the results in your `ex-3-4-2.trans` file.

3. *Explain in your own words what the errors are that HOL detects. Include your answers as comments in your source code.*

# Functions in ML

---

Functions are central to functional programming languages such as ML, and to the HOL theorem prover. In HOL, inference rules are functions written in ML that return theorems as values. Beginners using HOL will mostly use pre-defined functions. Sophisticated users are able to define their own functions to develop specialized theories or proofs. Many times, these specializations automate laborious tasks or make problems of scale manageable.

Regardless of whether you are a beginner or a power user of HOL, a little bit of functional programming goes a long way in HOL. This chapter is not a comprehensive treatment of function programming in ML. However, the subset of ML we present here is sufficient to do most everything you will need to do in HOL.

One feature of ML is this: ML is a higher-order language which means functions are first-class objects, i.e., functions are allowable arguments to functions, and functions are able to return functions as values. This degree of expressiveness provides a degree of clarity and succinctness unrivaled by languages that are not higher-order.

## 4.1 Functions, Evaluation Rules, and Values

**Functional abstraction and function application:** As a simple introduction to the notation, logic of functions, how they are evaluated, and functions as values in ML and HOL, consider the following expression:

$$x + 1,$$

where  $x$  is a variable that stands for any integer, and the value returned by the function is the result of adding 1 to the value of  $x$ . The value of the expression  $x + 1$  depends on the value of  $x$ . For example, if the value of  $x$  is 2, then the value of  $x + 1$  is 3.

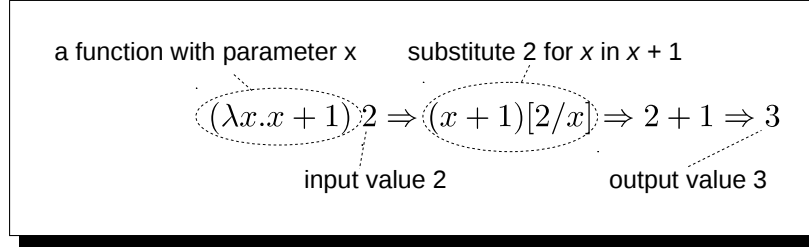
Consider the expression below whose value is a function that takes an integer input  $x$  and returns an output whose value is one more than the value of  $x$ .

$$\lambda x. (x + 1)$$

The above expression says “give me an argument  $x$  and I’ll give you the value  $x + 1$ .”  $\lambda v. e$  binds free occurrences of variable  $v$  in expression  $e$  that are not within the scope of another instance of  $\lambda v$ , so that when a value is substituted for  $v$ , that value is substituted into all free occurrences of  $v$ . The following shadow box shows how the function  $\lambda x. x + 1$  evaluates to 3 when applied to the value 2. There are three steps.

1. The function  $\lambda x. x + 1$  is applied to input value 2,
2. 2 is substituted into  $x$  within the expression  $x + 1$ , and
3.  $2 + 1$  is evaluated to produce the output 3.

The above is written symbolically as (where  $\Rightarrow$  is “evaluates to”),



Generalizing the above with variable  $x$ , and expressions  $e, e_1$ , and  $e_2$ , we have the following patterns and terminology:

$v$	a <b>variable</b>
$(\lambda v. e)$	functional <b>abstraction</b>
$(e_1 e_2)$	function <b>application</b> *
*function application is <b>left associative</b> , i.e., $e_1 e_2 e_3 = (e_1 e_2) e_3$	

The inference rule defining function application is known as *Beta Reduction* or *Beta Conversion* in the lambda calculus [Paulson, 1996] and [HOL, 2015b]—the name given to reasoning about computation using function abstraction and application. The rule is:

$$\text{Beta Conversion} \quad \frac{}{(\lambda x. e_1) e_2 = e_1[e_2/x]}$$

where  $e_1[e_2/x]$  is the result of substituting  $e_2$  for unbound instances of  $x$  in expression  $e_1$ , and substitution results in renaming of variables to prevent free (i.e., unbound) variables in  $e_2$  from being bound after substitution into  $e_1$ .

The following examples illustrate functional abstraction and function application.

#### Example 4.1

Suppose we wish to define a function that doubles  $x$  and adds it to  $y$  using *functional abstraction*. We do the following.

1. We start off with an expression stating the calculation  $2x + y$ .
2. We note that  $x$  and  $y$  are variables into which values are substituted.
3. Given the form of functional abstraction we introduced above, we have two ways expressing the function using functional abstraction:
  - (a) Variable  $x$  is the first bound variable into which values are substituted followed by  $y$ . This is shown in the shadow box below.

$$\lambda x. (\lambda y. (2x + y))$$

- (b) Variable  $y$  is the first bound variable into which values are substituted followed by  $x$ . This is shown in the shadow box below.

$$\lambda y.(\lambda x.(2x + y))$$

Both functional abstractions are acceptable. Nonetheless, the order in which variables appear, e.g.,  $\lambda x.\lambda y$  as opposed to  $\lambda y.\lambda x$  does matter in terms of the behavior of function application. We see this in the next two examples.  $\diamond$

#### Example 4.2

This example looks at the details of determining the value of a *function application* where the function is defined using functional abstraction. Suppose we wish to determine the value of  $(\lambda x.(\lambda y.2x + y)) 1 2$ .

The above expression has the form  $e_1 e_2 e_3$ , where  $e_1 = \lambda x.(\lambda y.2x + y)$ ,  $e_2 = 1$ , and  $e_3 = 2$ . Recalling that function application is *left associative*, we add parentheses as follows to make the order of evaluation explicit  $((\lambda x.(\lambda y.2x + y)) 1) 2$ . In other words (1) we evaluate  $(\lambda x.(\lambda y.x + y)) 1$ —call that value  $e$ , then (2) we evaluate the function application  $e 2$ . The following derivation is based on the *Beta Conversion* rule, substitution, arithmetic, and substituting equals for equals.

1.	$((\lambda x.(\lambda y.(2x + y))) 1) 2$	$= ((\lambda y.(2x + y))[1/x]) 2$	Beta Conversion: $((\lambda x.(\lambda y.(2x + y))) 1) = (\lambda y.(2x + y))[1/x]$
2.		$= (\lambda y.(2 + y)) 2$	Substitution: $(\lambda y.(2x + y))[1/x] = \lambda y.(2 + y)$
3.		$= (2 + y)[2/y]$	Beta Conversion: $(\lambda y.(2 + y)) 2 = (2 + y)[2/y]$
4.		$= 2 + 2$	Substitution: $(2 + y)[2/y] = 2 + 2$
5.		$= 4$	Arithmetic: $2 + 2 = 4$

The above derivation justifies the statement that the value of  $(\lambda x.(\lambda y.2x + y)) 1 2$  is 4.  $\diamond$

#### Example 4.3

Suppose we wish to determine the value of  $(\lambda y.(\lambda x.2x + y)) 1 2$ , where the variable ordering is reversed from the previous functional abstraction.

Just as before, the following derivation is based on Beta Conversion, substitution, arithmetic, and substituting equals for equals.

1.	$((\lambda y.(\lambda x.(2x + y))) 1) 2$	$= ((\lambda x.(2x + y))[1/y]) 2$	Beta Conversion: $((\lambda y.(\lambda x.(2x + y))) 1) = (\lambda x.(2x + y))[1/y]$
2.		$= (\lambda x.(2x + 1)) 2$	Substitution: $(\lambda x.(2x + y))[1/y] = \lambda x.(2x + 1)$
3.		$= (2x + 1)[2/x]$	Beta Conversion: $(\lambda x.(2x + 1)) 2 = (2x + 1)[2/x]$
4.		$= 2 \times 2 + 1$	Substitution: $(2x + 1)[2/x] = 2 \times 2 + 1$
5.		$= 5$	Arithmetic: $2 \times 2 + 1 = 5$

The above derivation justifies the statement that the value of  $(\lambda y.(\lambda x.2x + y)) 1 2$  is 5.  $\diamond$

**Returning functions as values:** In our introduction to functions, we said that functions are first-class objects, i.e., functions are able to return functions as values. In Examples 4.1 and 4.3, we saw examples of a function applied to an argument returning a function as a result. Specifically,

$$\begin{aligned} (\lambda x. (\lambda y. (2x + y))) 1 &= \lambda y. (2 + y) \\ (\lambda y. (\lambda x. (2x + y))) 1 &= \lambda x. (2x + 1) \end{aligned}$$

What the above shows is when a function of two variables  $x$  and  $y$  is applied to its first argument, the result is a *function of a single variable*,  $y$  for the first case and  $x$  for the second. This is an instance of function application returning a function as a value.

In the next section, we look at how functional abstraction and function application work in ML.

## 4.2 Functions, Evaluation Rules and Values in ML

ML is limited to the ASCII character set. Generally, a  $\lambda$ -expression  $\lambda v.e$  is written in ML as `fn v => e`. For the specific case  $\lambda x. (\lambda y. (2x + y))$ , the corresponding ML expression is `fn x => (fn y => 2*x + y)`.

In the next example, we replicate everything we did in Examples 4.2, 4.1, and 4.3 in ML.

### Example 4.4

Do the following:

1. Create a file `mlLab3.sml` in the subdirectory of ML labs.
2. Open the empty file in Emacs.
3. Add to `mlLab3.sml` what is shown in the shadow box.

```
(*****
(* mlLab3.sml                                     *)
(* Author: <your name>                             *)
(* Date: <today's date>                             *)
(*****
(fn x => (fn y => 2*x + y)) 1 2;
(fn y => (fn x => 2*x + y)) 1 2;
(fn x => (fn y => 2*x + y)) 1;
(fn y => (fn x => 2*x + y)) 1;
```

4. Open a HOL session within Emacs either with [M-h 2](#), [M-h 3](#), or using the corresponding menu buttons `HOL` `Process` `Start HOL-vertical split` or `Start HOL-horizontal split`.
5. Copy each expression into the HOL interpreter and evaluate it.
6. What you should see is in the numbered session box below.

```

- (fn x => (fn y => 2*x + y)) 1 2;
> val it = 4 : int
- (fn y => (fn x => 2*x + y)) 1 2;
> val it = 5 : int
- (fn x => (fn y => 2*x + y)) 1;
> val it = fn : int -> int
- (fn y => (fn x => 2*x + y)) 1;
> val it = fn : int -> int

```

1

The interpretations of the above results is as follows:

1. The value of expression  $(\text{fn } x \Rightarrow (\text{fn } y \Rightarrow 2x + y)) 1 2$  is the integer 4.
2. The value of expression  $(\text{fn } y \Rightarrow (\text{fn } x \Rightarrow 2x + y)) 1 2$  is the integer 5.
3. The value of expression  $(\text{fn } x \Rightarrow (\text{fn } y \Rightarrow 2x + y)) 1$  is a function that takes an integer as input and returns an integer. This function corresponds to  $\lambda y.(2 + y)$ .
4. The value of expression  $(\text{fn } y \Rightarrow (\text{fn } x \Rightarrow 2x + y)) 1$  is a function that takes an integer as input and returns an integer. This function corresponds to  $\lambda x.(2x + 1)$ .  $\diamond$

### 4.3 Naming Functions in ML

**Naming functions using `val` in ML:** Example 4.4 above shows functions are first-class objects in ML. The expressions  $(\text{fn } x \Rightarrow (\text{fn } y \Rightarrow 2x + y)) 1$  and  $(\text{fn } y \Rightarrow (\text{fn } x \Rightarrow 2x + y)) 1$  returned functions as values corresponding to  $\lambda y.(2 + y)$  and  $\lambda x.(2x + 1)$ , respectively.

Recall in Section 3.3 that constant names are given their associated values in ML with the declaration `val name = expression`. As shown in Example 4.4, functional abstractions are ML expressions and can be given a name.

#### Example 4.5

Do the following:

1. Add to *mlLab3.sml* what is shown in the shadow box.

```

(*****)
(* Naming functions defined using functional abstraction *)
(*****)
val fun1 = (fn x => (fn y => 2*x + y));
val fun2 = (fn y => (fn x => 2*x + y));
fun1 1 2;
fun2 1 2;
fun1;
fun2;
fun1 1;
fun2 1;
val fun3 = fun1 1;
val fun4 = fun2 1;
fun3 2;
fun4 2;

```

2. Copy each expression into the HOL interpreter and evaluate it.

3. What you should see is in the numbered session box below.

2

```

- val fun1 = (fn x => (fn y => 2*x + y));
> val fun1 = fn : int -> int -> int
- val fun2 = (fn y => (fn x => 2*x + y));
> val fun2 = fn : int -> int -> int
- fun1 1 2;
> val it = 4 : int
- fun2 1 2;
> val it = 5 : int
- fun1;
> val it = fn : int -> int -> int
- fun2;
> val it = fn : int -> int -> int
- fun1 1;
> val it = fn : int -> int
- fun2 1;
> val it = fn : int -> int
- val fun3 = fun1 1;
> val fun3 = fn : int -> int
- val fun4 = fun2 1;
> val fun4 = fn : int -> int
- fun3 2;
> val it = 4 : int
- fun4 2;
> val it = 5 : int

```

The interpretation of the above results in ML is as follows.

1. The result of the function declaration `val fun1 = (fn x => (fn y => 2*x + y))` is `val fun1 = fn : int -> int -> int`. What ML is saying is that the name `fun1` is a function (as indicated by `fn`) whose type signature is `int -> int -> int`. In type signatures  $\rightarrow$  is *right associative*, i.e.,  $int \rightarrow int \rightarrow int = int \rightarrow (int \rightarrow int)$ . The type of `fun1` (with parentheses added for clarity) is: `int -> (int -> int)`. Specifically, when `fun1` is applied to an `int`, the value returned is a function whose type is `int -> int`, i.e., another function that when applied to an `int` returns an `int`.
2. The result of the function declaration `val fun2 = (fn y => (fn x => 2*x + y))` has the corresponding interpretation for the declaration defining `fun1`.
3. The result of the function application `fun1 1 2` corresponding to  $(\lambda x.(\lambda y.2x+y))\ 1\ 2$  is 4, as expected.
4. The result of the function application `fun2 1 2` corresponding to  $(\lambda y.(\lambda x.2x+y))\ 1\ 2$  is 5, as expected.
5. The value and type signature of `fun1` is `val it = fn : int -> int -> int`.
6. The value and type signature of `fun2` is `val it = fn : int -> int -> int`.
7. The result of function application `fun1 1` is a function `val it = fn : int -> int`.
8. The result of function application `fun2 1` is a function `val it = fn : int -> int`.
9. The result of declaring `val fun3 = fun1 1` is a function corresponding to  $\lambda y.(2+y)$ .
10. The result of declaring `val fun4 = fun2 1` is a function corresponding to  $\lambda x.(2x+1)$ .



11. The result of the function application `fun3 2` is `val it = 4 : int`, i.e., the integer 4, corresponding to  $(\lambda y.(2+y))\ 2 = 4$ .
12. The result of the function application `fun4 2` is `val it = 5 : int`, i.e., the integer 5, corresponding to  $(\lambda x.(2x+1))\ 2 = 5$ . ◇

**Defining named functions using `fun` in ML:** Often, we will be defining and naming functions. To make defining named functions convenient, `fun` is used in ML to declare a function's name, variables (named in order), and the expression defining the behavior of the function. As an illustration, we define functions `exFun1` and `exFun2` using `fun` corresponding to `fun1` and `fun2` in Example 4.5.

#### Example 4.6

Do the following:

1. Add to `mLab3.sml` what is shown in the shadow box.

```
(*****
(* Defining functions using fun                                     *)
*****)
fun exFun1 x y = 2*x + y;
fun exFun2 y x = 2*x + y;
exFun1 1 2;
exFun2 1 2;
```

2. Copy each expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- fun exFun1 x y = 2*x + y;
> val exFun1 = fn : int -> int -> int
- fun exFun2 y x = 2*x + y;
> val exFun2 = fn : int -> int -> int
- exFun1 1 2;
> val it = 4 : int
- exFun2 1 2;
> val it = 5 : int
```

3

The results above correspond directly to the definitions of `fun1` and `fun2` in Example 4.5. ◇

## 4.4 Defining Functions Using Pattern Matching in ML

Using pattern matching in function definitions often results in definitions that are clear and concise. Suppose we wish to define a function that takes a pair of integers and sums the square of each element. Specifically,

$$\text{pairSquareSum}(e_1, e_2) = (e_1 \times e_1) + (e_2 \times e_2).$$

The following example shows the definition of `pairSquareSum` in ML.

**Example 4.7**

Do the following.

1. Add to *mLab3.sml* what is shown below in the shadow box.

```
(*****
(* Defining functions using pattern matching *)
*****)
fun pairSquareSum (e1,e2) = (e1 * e1) + (e2 * e2);
pairSquareSum (2,3);
```

2. Copy each expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- fun pairSquareSum (e1,e2) = (e1 * e1) + (e2 * e2);
> val pairSquareSum = fn : int * int -> int
- pairSquareSum (2,3);
> val it = 13 : int
```

4

The result in the last line corresponds to  $(\lambda(e_1, e_2).((e_1 \times e_1) + (e_2 \times e_2)))(2, 3) = 2^2 + 3^2 = 4 + 9 = 13$ .  $\diamond$

Suppose we want a function *pairSquareSumList* that takes a list of pairs of integers and returns a list of integers corresponding to the sum of the squares of the pairs. Recall, lists have two forms: (1) the empty list  $[]$ , and (2) the non-empty list *head* :: *tail*, where *head* is the first element of a non-empty list, :: is the list constructor, and *tail* is the rest of the non-empty list. As a reminder, the list  $[1, 2, 3]$  is the same as  $1 :: [2, 3]$ , i.e., the integer 1 pushed onto the list of integers  $[2, 3]$ . The list with a single element  $[(true, false)]$  is the same as  $(true, false) :: []$ , i.e., the pair of booleans  $(true, false)$  pushed onto the empty list of pairs of booleans  $[]$ .

When we define functions on lists, often our definitions will have two parts: (1) the result when the function is applied to an empty list of the form  $[]$ , and (2) the result when the function is applied to a non-empty list of the form  $x :: xs$ . The following definition of *pairSquareSumList* uses pattern matching.

$$\begin{aligned} \text{pairSquareSumList } [] &= [] \\ \text{pairSquareSumList } (x :: xs) &= (\text{pairSquareSum } x) :: (\text{pairSquareSumList } xs) \end{aligned}$$

Notice that the above definition is *recursive*, i.e., *pairSquareSumList* is defined using itself. Informally, we observe that the recursion will terminate because for non-empty lists  $x :: xs$ , *pairSquareSumList* is applied to *xs*, which is a smaller list than  $x :: xs$  by one element. All recursive calls end with *pairSquareSumList*  $[] = []$ . The following derivation shows the result of applying the above definition of *pairSquareSumList* to  $[(2, 3)]$ , the list containing the single pair  $(2, 3)$ .

1.	$\text{pairSquareSumList } [(2, 3)] = \text{pairSquareSumList } ((2, 3) :: [])$	Rewrite $[(2, 3)]$ as $(2, 3) :: []$
2.	$= (\text{pairSquareSum } (2, 3)) :: (\text{pairSquareSumList } [])$	Apply $\text{pairSquareSumList } (x :: xs) = (\text{pairSquareSum } x) :: (\text{pairSquareSumList } xs)$
3.	$= 13 :: []$	Apply $\text{pairSquareSum } (2, 3) = 13$ and $\text{pairSquareSumList } [] = []$
4.	$= [13]$	Rewrite $13 :: []$ as $[13]$

The following example shows how the above definition of *pairSquareSumList* is defined in ML. Note the use of  $|$  to separate the two definitions corresponding to the patterns of empty and non-empty lists.

**Example 4.8**

Do the following.

1. Add to *mLab3.sml* what is shown below in the shadow box.

```
fun pairSquareSumList [] = []
  | pairSquareSumList (x::xs) = (pairSquareSum x) :: pairSquareSumList xs;

pairSquareSumList [(2,3)];
pairSquareSumList [(0,1), (2,3)];
```

2. Copy each expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- fun pairSquareSumList [] = []
  | pairSquareSumList (x::xs) = (pairSquareSum x) :: pairSquareSumList xs;
> val pairSquareSumList = fn : (int * int) list -> int list
- pairSquareSumList [(2,3)];
> val it = [13] : int list
- pairSquareSumList [(0,1), (2,3)];
> val it = [1, 13] : int list
```

5

The above shows that the type of `pairSquareSumList` takes a list of pairs of integers and returns a list of integers. The value of `pairSquareSumList [(2,3)]` is the same as what we derived before. The value of `pairSquareSumList [(0,1), (2,3)]` is `[1, 13]`, corresponding to the list  $[(0^2 + 1^2), (2^2 + 3^2)]$ .  $\diamond$

## 4.5 Let Expressions in ML

When making value declarations or defining functions, sometimes we may make use of *helper* declarations in the form of *local* declarations within `let` expressions in ML. The general form of `let` expressions in ML is:

`let  $D$  in  $E$  end`

where  $D$  is a sequence of declarations using `val` or `fun`, and  $E$  is an expression. If  $D$  consists of a sequence of declarations, then each declaration is seen by the remaining declarations.

**Example 4.9**

This example shows how we can make local declaration of values that are not global assignments of values to constant names. Suppose we wish to compute the value of  $x + y + z$ , where  $x = 1$ ,  $y = 2$ , and  $z = 3$ . For whatever reason, we do not need to make  $x$ ,  $y$ , or  $z$  global declarations.

Do the following.

1. Add to *mLab3.sml* what is shown below in the shadow box.

```

(*****
(* Let expressions *)
(*****
(* x, y, and z do not have any values assigned to them. *)
(*****
x;
y;
z;
val letExample1 =
let
  val x = 1
  val y = 2
  val z = 3
in
  x + y + z
end;
x;
y;
z;

```

2. Copy each expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```

- x;
! Toplevel input:
! x;
! ^
! Unbound value identifier: x
- y;
! Toplevel input:
! y;
! ^
! Unbound value identifier: y
- z;
! Toplevel input:
! z;
! ^
! Unbound value identifier: z
- val letExample1 =
let
  val x = 1
  val y = 2
  val z = 3
in
  x + y + z
end;
> val letExample1 = 6 : int
- x;
! Toplevel input:
! x;
! ^
! Unbound value identifier: x
- y;
! Toplevel input:
! y;
! ^
! Unbound value identifier: y
- z;

```

6

The above session shows the computation of  $x + y + z$  for a local assignment of integer values 1, 2, and 3,

respectively to  $x$ ,  $y$ , and  $z$ . The “*Unbound value identifier*” errors surrounding the `let` expression indicate that the assignment of values to  $x$ ,  $y$ , and  $z$  is only within the scope of the `let` expression. ◇

#### Example 4.10

Suppose we wish to define the function `sumSquaredList`, which takes a list of integers, squares them, then adds them together. Do the following.

1. Add to `mLab3.sml` what is shown below in the shadow box.

```
(*****
 * Definition of squareSumList
 *****)
fun squareSumList intList =
let
  fun square x = x * x
  fun sumList [] = 0
    | sumList (x::xs) = x + sumList xs
in
  square(sumList intList)
end;

squareSumList [1,2,3];
```

2. Copy each expression into the HOL interpreter and evaluate it.
3. What you should see is in the numbered session box below.

```
- fun squareSumList intList =
let
  fun square x = x * x
  fun sumList [] = 0
    | sumList (x::xs) = x + sumList xs
in
  square(sumList intList)
end;
> val squareSumList = fn : int list -> int
- squareSumList [1,2,3];
> val it = 36 : int
```

7

The definition of `squareSumList` includes the local definitions of `square` and `sumList`. The definition of `sumList` is recursive. For clarity, we derive the value of `sumList [1,2,3]` using the definition of `sumList` in the `let` expression.

1. $\text{sumList } [1,2,3] = \text{sumList } (1 :: [2,3])$	Rewrite with $[1,2,3] = (1 :: [2,3])$
2. $= 1 + \text{sumList } [2,3]$	Rewrite with $\text{sumList } (1 :: [2,3]) = 1 + \text{sumList } [2,3]$
3. $= 1 + (\text{sumList } (2 :: [3]))$	Rewrite with $[2,3] = 2 :: [3]$
4. $= 1 + (2 + \text{sumList } [3])$	Rewrite with $\text{sumList } (2 :: [3]) = 2 + \text{sumList } [3]$
5. $= 1 + (2 + \text{sumList } (3 :: []))$	Rewrite with $[3] = (3 :: [])$
6. $= 1 + (2 + (3 + \text{sumList } []))$	Rewrite with $\text{sumList } (3 :: []) = 3 + \text{sumList } []$
7. $= 1 + (2 + (3 + 0))$	Rewrite with $\text{sumList } [] = 0$
8. $= 6$	Rewrite with $1 + (2 + (3 + 0)) = 6$

◇

## 4.6 Exercises

**Exercise 4.6.1** Compute the value of the following function applications using Beta Conversion and substitution.

- A.  $((\lambda x. (\lambda y. (4x + y))) 4) 1$
- B.  $((\lambda y. (\lambda x. (4x + y))) 4) 1$
- C.  $((\lambda a. (\lambda b. (a :: b))) (1, true)) [(0, false)]$
- D.  $((\lambda p. (\lambda q. (p, q))) true) [1, 2, 3]$
- E.  $((\lambda p. (\lambda q. (p, q))) [1, 2, 3]) true$
- F.  $(\lambda (a, b). (a + b)) (1, 2)$

**Exercise 4.6.2** Use functional abstraction to define functions with the following behavior.

- A. A function that takes a 3-tuple of integers  $(x, y, z)$  as input and returns the value corresponding to the sum  $x + y + z$ .
- B. A function that takes two integer inputs  $x$  and  $y$  (where  $x$  is supplied first followed by  $y$ ) and returns the boolean value corresponding to  $x < y$ .
- C. A function that takes two strings  $s_1$  and  $s_2$  (where  $s_1$  is supplied first followed by  $s_2$ ) and concatenates them, where “ $\wedge$ ” denotes string concatenation. For example, “Hi”  $\wedge$  “ there” results in the string “Hi there”.
- D. A function that takes two lists  $list_1$  and  $list_2$  (where  $list_1$  comes first) and appends them, where “ $@$ ” denotes list append. For example  $[true, false] @ [false, false, false]$  results in the list  $[true, false, false, false, false]$ .
- E. A function that takes a pair of integers  $(x, y)$  and returns the larger of the two values. You note that the conditional statement `if condition then a else b` returns `a` if `condition` is true, otherwise it returns `b`.

**Exercise 4.6.3** In ML, define functions (and test them on examples) corresponding to each function in Exercise 4.6.2. For each of the functions in Exercise 4.6.2, you will define **two** ML functions, (1) the first using `fn` and `val` to define and name the function, and (2) the other using `fun` to define and name the function. Make sure you use pattern matching.

For example, suppose the function is  $\lambda x. (\lambda y. 2x + y)$ . We would define in ML

1. `val funEx1 = (fn x => (fn y => 2*x + y)), and`
2. `fun funEx2 x y = 2*x + y`

As a naming convention, use the names `funA1`, `funA2`, `funB1`, `funB2`, etc.

**Exercise 4.6.4** In ML, define a function `listSquares` that when applied to the empty list of integers returns the empty list, and when applied to a non-empty list of integers returns a list where each element is squared. For example, `listSquares [2, 3, 4]` returns `[4, 9, 16]`. Define the function using a `let` expression in ML.

# Polymorphism and Higher-Order Functions in ML

Pattern matching provides succinctness, preciseness, and elegance to function definitions. Example 5.1 shows the definition of a function that takes a pair of integers and returns the sum of each element of the pair.

## Example 5.1

Do the following:

1. Create a new file *mlLab4.sml* and enter what is shown below in the shadow box.

```
(*****)
(* mlLab4.sml *)
(* Author: <your name> *)
(* Date: <today's date> *)
(*****)
fun sumSquare(x,y) =
let
  fun square x = x * x
in
  (square x) + (square y)
end;

sumSquare (2,3);
```

2. Start up HOL within Emacs and copy each expression into the HOL interpreter for evaluation.
3. What you should see is in the numbered session box below.

```
1
- fun sumSquare(x,y) =
let
  fun square x = x * x
in
  (square x) + (square y)
end;
> val sumSquare = fn : int * int -> int
- sumSquare (2,3);
> val it = 13 : int
```

The result  $2^2 + 3^2 = 13$  is what we expect given the definition of `sumSquare` and the local definition of `square`. ◇

Looking at the definition of `sumSquare` in Example 5.1 and its type signature in the above session box, how did the ML interpreter within HOL derive the type of `sumSquare` as `int * int -> int`, when the definition of `sumSquare` contained no explicit type information?

The answer is ML and HOL are able to *infer types* based on the terms within expressions. In other words, the logical rules for inferring types is built into ML and HOL. In the specific case of `sumSquare`, the use of the infix operators `*` and `+` allow ML to infer that `x`, `y`, and the output value must be of type `int`.

We already saw evidence of type inference in Section 3.1 with type clashes. Example 5.2 illustrates ML's capability to detect type inconsistencies within expressions.

### Example 5.2

Do the following.

1. Add to `mLab4.sml` what is shown below in the shadow box.

```
(*****
(* Type inference allows ML to detect type clashes *)
*****)
1 + "1";
1 + [2];
```

2. Evaluate each expression within the HOL interpreter.
3. What you should see is in the numbered session box below.

```
- 1 + "1";
! Toplevel input:
! 1 + "1";
!   ^^
! Type clash: expression of type
!   string
! cannot have type
!   int
- 1 + [2];
! Toplevel input:
! 1 + [2];
!   ^^
! Type clash: expression of type
!   'a list
! cannot have type
!   int
```

2

In both cases above, the second argument to `+` is inconsistently typed. In the first case, ML complains that strings are not integers. In the second case, ML complains that the integer list `[1]` is not an integer. ◇

Pattern matching and type checking provide useful benefits in the form succinctness and correctness. In the next section, we introduce polymorphism, which provides flexibility and generality to types and functions.

## 5.1 Polymorphism

*Polymorphism* (having many forms) in ML takes the form of *type variables*, where type variables may be associated with any type. The use of variables in type signatures supports *pattern matching in type signatures*. Type variables provide generality, flexibility, and applicability to functions in ML. Example 5.3 illustrates the use and benefits of type variables and polymorphism.



**Example 5.3**

We devise a function `mkpair` that takes two arguments of any type and creates a pair, where the first element of the pair is the first argument and the second element of the pair is the second argument. Do the following.

1. Add to *mlLab4.sml* what is shown below in the shadow box.

```
(*****
(* A polymorphic function, where its arguments are any type          *)
*****)
fun mkpair a b = (a,b);

mkpair 1 2;
mkpair "true" false;
mkpair (1,true) ["a","b"];
```

2. Evaluate each expression within the HOL interpreter.
3. What you should see is in the numbered session box below.

```
- fun mkpair a b = (a,b);
> val ('a, 'b) mkpair = fn : 'a -> 'b -> 'a * 'b
- mkpair 1 2;
> val it = (1, 2) : int * int
- mkpair "true" false;
> val it = ("true", false) : string * bool
- mkpair (1,true) ["a","b"];
> val it = ((1, true), ["a", "b"]) : (int * bool) * string list
```

3

The result of evaluating `fun mkpair a b = (a,b)` yields

```
val ('a, 'b) mkpair = fn : 'a -> 'b -> 'a * 'b.
```

Types such as `'a` and `'b` are something we have not yet seen in ML. `'a` and `'b` are *type variables*, i.e., they stand for *any type*. Type variables in ML and HOL follow a syntactic convention: all type variables start with a prime or single quote character `'`. For example, `'thisIsATypeVariable` is a type variable.

What `val ('a, 'b) mkpair = fn : 'a -> 'b -> 'a * 'b` conveys is that the function `mkpair` has type parameters in the form of type variables `'a` and `'b`. The type signature of `mkpair` is it takes as its first argument a value of type `'a`, a second argument a value of type `'b`, and returns a pair of type `'a * 'b`.

The evaluation of `mkpair 1 2` and `mkpair "true" false` show the benefits of polymorphism. The *same function* `mkpair` in the first expression is applied to two integers and returns a pair of integers; in the second expression `mkpair` is applied to a string and a boolean and returns a string-boolean pair. By matching type variables in the type signature of `mkpair`, we see that the type of the first element of the resulting pair is the same type of the first argument and the type of the second element of the resulting pair is the same type as the second argument.

The evaluation of `mkpair (1,true) ["a","b"]` shows that type variables can be instantiated with compound types. In this case the result is the type variable `'a` is instantiated to type `int * bool`; type variable `'b` is instantiated to type `string list`. The result `((1, true), ["a", "b"])` is of type `(int * bool) * string list`.

Finally, the expressions `mkpair:'tvar1 -> 'tvar2 -> 'tvar1 * 'tvar2` and `mkpair:int -> bool -> int * bool` show how type variables (1) are renamed to other type variables, and (2) instantiated to specific types. ◇

Type variables and polymorphism enable the same function to be applied to different argument types. This avoids having to define multiple instances of the same function on different types.

## 5.2 Higher-Order Functions

Functions that take functions as inputs are higher-order functions. Higher-order functions are built-into ML; ML allows users to define higher-order functions.

**The higher-order function *map*:** *map* is one of the higher-order functions built-in ML. Informally, what *map f list* does is apply the function *f* to each element in *list*. For example, consider the function  $\lambda x.2x$ , which takes an integer *x* and doubles it, and the list  $[1, 2]$ . *map*  $(\lambda x.2x)$   $[1, 2]$  returns  $[2, 4]$ .

*map* is higher order because it takes functions as arguments. Using functions as parameters simplifies function definition. Suppose instead of doubling each element of a list of integers we wish to negate them. All we need to do is give *map* a function that negates integers, e.g.,  $\lambda x. -x$ . For example, the value of *map*  $(\lambda x. -x)$   $[1, 2]$  is  $[-1, -2]$ .

*map* is polymorphic. Instead of a list of integers, we could have given it a list of *pairs* of integers, e.g.,  $[(1, 2), (3, 4)]$ . In this case, the function we give to *map* must operate on pairs of integers. For example, suppose we wish to add each element in a list of integer pairs and return a list of their sums, e.g., with  $[(1, 2), (3, 4)]$  as input we want  $[3, 7]$  as output. To accomplish this using *map*, we can use the function *map*  $(\lambda(x, y).(x + y))$   $[(1, 2), (3, 4)]$ .

The formal definition of *map* is as follows.

$$\begin{aligned} \text{map } f [] &= [] \\ \text{map } f (x :: xs) &= ((f x) :: (\text{map } f xs)) \end{aligned}$$

### Example 5.4

We derive the value of *map*  $(\lambda x.2x)$   $[1, 2]$  using the formal definition of *map*.

- |    |   |  |
|----|---|--|
| 1. | $\text{map } (\lambda x.2x) [1, 2] = \text{map } (\lambda x.2x) (1 :: [2])$ | $[1, 2] = (1 :: [2])$  |
| 2. | $= ((\lambda x.2x) 1) :: (\text{map } (\lambda x.2x) [2])$                  | From definition of <i>map</i> , $\text{map } (\lambda x.2x) (1 :: [2]) = ((\lambda x.2x) 1) :: (\text{map } (\lambda x.2x) [2])$ |
| 3. | $= 2 :: (\text{map } (\lambda x.2x) [2])$                                   | From function application, $(\lambda x.2x) 1 = 2$  |
| 4. | $= 2 :: (\text{map } (\lambda x.2x) (2 :: []))$                             | $[2] = 2 :: []$  |
| 5. | $= 2 :: ((\lambda x.2x) 2) :: (\text{map } (\lambda x.2x) [])$              | From definition of <i>map</i> , $\text{map } (\lambda x.2x) (2 :: []) = ((\lambda x.2x) 2) :: (\text{map } (\lambda x.2x) [])$   |
| 6. | $= 2 :: (4 :: (\text{map } (\lambda x.2x) []))$                             | From function application $(\lambda x.2x) 2 = 4$   |
| 7. | $= 2 :: (4 :: [])$  | From definition of <i>map</i> , $\text{map } (\lambda x.2x) [] = []$   |
| 8. | $= [2, 4]$  | $2 :: (4 :: []) = 2 :: [4] = [2, 4]$   |

The above derivation show that *map* starts with the first element of a non-empty list, applies *f*, and recursively calls itself on the remainder of a non-empty list. It stops when it encounters the empty list.  $\diamond$

**map in ML:** *map* is one of the most common and frequently used higher-order function in functional programming. The following example shows how *map* works in ML.

**Example 5.5**

Do the following:

1. Add to *mlLab4.sml* what is shown below in the shadow box.

```
(*****
(* Higher-order functions in ML *)
*****)

(*****
(* Using map *)
*****)
map;
map (fn x => 2*x) [1,2];
map (fn x => ~x) [1,2];
map (fn (x,y) => x + y) [(1,2), (3,4)];
```

2. Evaluate each expression within the HOL interpreter.
3. What you should see is in the numbered session box below.

```
- map;
> val ('a, 'b) it = fn : ('a -> 'b) -> 'a list -> 'b list
- map (fn x => 2*x) [1,2];
> val it = [2, 4] : int list
- map (fn x => ~x) [1,2];
> val it = [~1, ~2] : int list
- map (fn (x,y) => x + y) [(1,2), (3,4)];
> val it = [3, 7] : int list
```

4

Evaluating *map* alone results in its type signature. We see that it is a *polymorphic* function whose first argument is a function with type signature *'a -> 'b*, and whose second argument is of type *'a list*. The output value is of type *'b list*. This matches exactly with the type of the input function.

The subsequent three evaluations of *map* applied to the three functions and lists matches our derivation in Example 5.4 and the previous informal descriptions. ◇

**The higher-order function filter:** Another frequently used higher-order function is *filter*. Its formal definition is as follows.

$$\begin{aligned} \text{filter } P [] &= [] \\ \text{filter } P (x :: xs) &= \text{if } P x \text{ then } x :: (\text{filter } P xs) \text{ else } (\text{filter } P xs) \end{aligned}$$

Informally, the *filter* function takes a testing function *P* and a list. If the list is empty, then *filter* returns the empty list. If the list is non-empty, *filter* returns a list where all elements *x* in the list satisfy *P*, i.e., *P x* is true. In other words, all elements *e* in the input list for which *P e* is false are filtered out of the input list.

**filter in ML:** *filter* is one of the higher-order functions built into ML. Its definition is the standard definition given above. Example 5.6 below shows the operation of *filter* in ML.

**Example 5.6**

Do the following:

1. Add to *mLab4.sml* what is shown below in the shadow box.

```
(*****
(* Using filter
*****)
filter;
filter (fn x => x < 5) [4,6];
filter (fn x => x < 5) [1,2,4,8,16];
```

2. Evaluate each expression within the HOL interpreter.
3. What you should see is in the numbered session box below.

```
- filter;
> val 'a it = fn : ('a -> bool) -> 'a list -> 'a list
- filter (fn x => x < 5) [4,6];
> val it = [4] : int list
- filter (fn x => x < 5) [1,2,4,8,16];
> val it = [1, 2, 4] : int list
```

5

The type signature of `filter` is very similar to that of `map`, except that type variable `'b` in `map` is replaced by `bool`. The reason is the function given to `filter` is a test, i.e., yields true or false as a result. The remaining results are

◇

**More examples:** In this example, we want a function `doubleAllLessThan n intList`. The function, when given an integer `n` and a list of integers `intList`, returns another list of integers where each element corresponds to doubling an element in `intList` that was less than `n`. There are several approaches to devising `doubleAllLessThan`. The approach we will take is to define locally the `double` function, and the intermediate list `helperList` as the filtered list using `n`. The result is obtained by mapping `double` over `helperList`.

The ML definition is shown below.

### Example 5.7

Do the following.

1. Add to *mLab4.sml* what is shown below in the shadow box.

```
(*****
(* Defining doubleAllLessThan
*****)
fun doubleAllLessThan n intList =
let
  fun double x = 2*x
  val helperList = filter (fn x => x < n) intList
in
  map double helperList
end;
```

2. Evaluate each expression within the HOL interpreter.
3. What you should see is in the numbered session box below.

```

- fun doubleAllLessThan n intList =
let
  fun double x = 2*x
  val helperList = filter (fn x => x < n) intList
in
  map double helperList
end;
> val doubleAllLessThan = fn : int -> int list -> int list
- doubleAllLessThan 10 [0,1,5,7,10,11,12];
> val it = [0, 2, 10, 14] : int list

```

6

Notice the local definitions of `double` and `helperList` within an `let` expression. ◇

## 5.3 Exercises

**Exercise 5.3.1** Derive the values of `map ( $\lambda x. -x$ ) [1,2]` is `[-1, -2]` and `map ( $\lambda(x,y).(x+y)$ )[(1,2),(3,4)]`. Use the derivation in Exercise 5.4 as your guide.

**Exercise 5.3.2** Define a function `Map` in ML, whose behavior is identical to `map`. Note: you cannot use `map` in the definition of `Map`. However, you can adapt the definition of `map` and use it in your definition. Show test cases of your function returning the expected results by comparing the outputs of both `Map` and `map`. Your examples should include the cases in Exercise 5.3.1.

**Exercise 5.3.3** Derive the values of `filter ( $\lambda x.x < 5$ )[4,6]` using the definition of `filter`. Your derivation should be similar to what you did in Exercise 5.3.1.

**Exercise 5.3.4** Define a function `Filter` in ML, whose behavior is identical to `filter`. Note: you cannot use `filter` in the definition of `Filter`. However, you can adapt the definition of `filter` and use it in your definition. Show test cases of your function returning the expected results by comparing the outputs of both `Filter` and `filter`. Your examples should include the cases in Exercise 5.3.3.

**Exercise 5.3.5** Define an ML function `addPairsGreaterThan n list`, whose behavior is defined as follows: (1) given an integer `n`, and (2) given a list of pairs of integers `list`, `addPairsGreaterThan n list` will return a list of integers where each element is the sum of integer pairs in `list` where both elements of the pairs are greater than `n`.

An example session is shown below.

```

- addPairsGreaterThan 0 [(0,1),(2,0),(2,3),(4,5)];
> val it = [5, 9] : int list

```

7

**Hint:** the logical operators for negation, conjunction, and disjunction ML are `not`, `andalso`, and `orelse`. This is shown in the following numbered session.

```

- not false;
> val it = true : bool
- false andalso true;
> val it = false : bool
- true andalso true;
> val it = true : bool
- false orelse false;
> val it = false : bool
- false orelse true;
> val it = true : bool

```

8

BLANK PAGE

## **Part III**

# **Lab Exercises: Introduction to HOL**





# Introduction to HOL

HOL (the Cambridge University Higher Order Logic proof checker) is implemented in ML. The relationship between ML and HOL is this: *HOL is implemented within ML as a set of types and functions that manipulate HOL objects*. More precisely, *HOL objects* are *HOL terms or formulas*, *HOL types*, and *HOL theorems*. HOL objects are manipulated by ML functions. ML is the *meta-language* that constructs and deconstructs HOL objects, and implements inference rules as functions returning theorems as values.

The primary capabilities for you to acquire in this first lab on HOL are:


- translating logical formulas into HOL's ASCII notation,
- turning on and off HOL's various printing switches that ASCII, display types, and assumptions
- practice dealing with HOL's type checker

## 6.1 HOL Terms

### 6.1.1 Entering HOL terms and types

The first essential capability you need is to enter HOL terms (formulas) using ML, comprehend what the system is telling you, display the types of HOL terms, and control what you want to see in emacs.

Please do the following:

1. In emacs, create and open a new file `holLab2.sml` within the subdirectory `HOL/HOLLab2`.
2. Start a HOL process by clicking `HOL`, `Process`, and either `Start HOL horizontal` or `Start HOL vertical`.
3. In the `*HOL*` window, type ``x`;` .

The result of doing the above is below.

1

```
-----
HOL-4 [Kananaskis 8 (stdknl, built Fri Aug 24 01:39:57 2012)]

For introductory HOL help, type: help "hol";

-----

[loading theories and proof tools ..... ]
[closing file "/usr/local/share/HOL/tools/end-init-boss.sml"]
- - - - - ``x``;
<<HOL message: inventing new type variable names: 'a>>
> val it = ``x`` : term
-
```

We have entered our first HOL term, but what does it mean? In particular, what does the HOL message about inventing new type variable name `'a` mean? What does `val it = ``x`` : term` mean?

A general rule in using HOL is this: **when in doubt look at the types of terms explicitly**. To activate displaying of types, do the following clicks: `HOL`, `Printing switches`, and `Show types`.

We re-enter the same term and look at the result. What you see is the following:

```
> val it = ``(x:α)`` : term
```

What the above means is this:

1. The value of the expression typed into ML, namely ```x```, is itself ```x```. The ML type of ```x``` is the **ML type** `term`.
2. The above shows that **HOL terms are surrounded by double back quotes** and HOL terms in ML have type `term`. You may think of this as all well-formed HOL formulas are defined in ML as ML type `term`.
3. Looking inside the double back quotes we see `x : α`. What this signifies is that within HOL, the type of variable `x` is `α`, namely a **type variable**. It is HOL's way of saying `x` is of any type, so let it be `α`. This corresponds to `x :: a` in Haskell.

The printing of `α` illustrates something else, HOL takes advantage of the capability to display characters not in the ASCII “typewriter” character set. This is *unicode*. While pleasant to look at, the question is *how do we type α on our QWERTY keyboard?*

To see the answer, do the following clicks: `HOL`, `Printing switches`, `Unicode`. Doing so toggles unicode display on and off.

Type in ```x```; `↵` again and see what happens. What we get is as follows.

```
- ** Unicode trace now off
- ``x``;
<<HOL message: inventing new type variable names: 'a>>
> val it = ``(x : 'a)`` : term
```

2

Now we can see fully what is going on. When we entered in ```x``` with no HOL type information, HOL did the best it could and concluded that `x` could be any type, so it called it `:'a`, which is pretty-printed as `: α`. (And yes, `:'b` is `: β`, `:'d` is `: δ`, etc.)

Now, type in ```P x``` and look at what happens.

```
- ``P x``;
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it =
  ``(P : 'a -> 'b) (x : 'a)``
  : term
```

3

Again, we have given HOL no information about any types, so it inferred that `P` is a function with type signature `:'a -> 'b`, i.e., `: α → β`.

When there is enough context, HOL can infer types on its own. Type in ```P x ==> y```; `↵`, which stands for `P(x) ⊃ y`, i.e., `P(x) implies y`. What you get follows and shows that HOL inferred ```x : 'a```, which implies that ```P : 'a -> bool``` because it is the antecedent of a logical implication ```==>```. For similar reasons, HOL infers that the type of `y` is ```: bool```.

```

- ``P x``;
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it =
  `` (P : 'a -> 'b) (x : 'a) ``
  : term
- ``P x ==> y``;
<<HOL message: inventing new type variable names: 'a>>
> val it =
  `` (P : 'a -> bool) (x : 'a) ==> (y : bool) ``
  : term

```

Of course, we can always explicitly specify the types of terms, much as we do in Haskell. Type in ```(f : num -> bool) (x:num) \ / (F:bool) ``; ↵`. What we get is shown below.

```

- ``(f:num -> bool) (x:num) \ / (F:bool) ``;
> val it =
  ``(f :num -> bool) (x :num) \ / F``
  : term

```

Of course, when we explicitly specify what amounts to conflicting type information, HOL complains. Type in ```(P : 'a -> bool) (x:num) ``; ↵`.

```

- ``(P : 'a -> bool) (x:num) ``;

Type inference failure: unable to infer a type for the application of

(P : 'a -> bool)

on line 30, characters 3-6

to

(x :num)

on line 30, characters 17-22

unification failure message: unify failed
! Uncaught exception:
! HOL_ERR

```

To do today's lab problems, please use the notation in Table 6.1.

## 6.2 Exercises

**Exercise 6.2.1** *In the following problems, enable HOL's Show types capability and disable Unicode so only ACSII characters are displayed.*

1. Enter the HOL equivalent of  $P(x) \supset Q(y)$ . Show what HOL returns. What are the types of  $x$ ,  $y$ ,  $P$ , and  $Q$ ?
2. Consider again  $P(x) \supset Q(y)$ . Suppose we wish to constrain  $x$  to HOL type `:num` and  $y$  to HOL type `:bool`. Re-enter your expression corresponding to  $P(x) \supset Q(y)$  and show that the types of  $x$ ,  $y$ ,  $P$ , and  $Q$  are appropriately typed.

Terms of the HOL Logic			
<i>Kind of term</i>	<i>HOL notation</i>	<i>Standard notation</i>	<i>Description</i>
Truth	$\top$	$\top$	<i>true</i>
Falsity	$\bot$	$\perp$	<i>false</i>
Negation	$\sim t$	$\neg t$	<i>not t</i>
Disjunction	$t_1 \vee t_2$	$t_1 \vee t_2$	<i>t<sub>1</sub> or t<sub>2</sub></i>
Conjunction	$t_1 \wedge t_2$	$t_1 \wedge t_2$	<i>t<sub>1</sub> and t<sub>2</sub></i>
Implication	$t_1 \Rightarrow t_2$	$t_1 \Rightarrow t_2$	<i>t<sub>1</sub> implies t<sub>2</sub></i>
Equality	$t_1 = t_2$	$t_1 = t_2$	<i>t<sub>1</sub> equals t<sub>2</sub></i>
$\forall$ -quantification	$\forall x. t$	$\forall x. t$	<i>for all x : t</i>
$\exists$ -quantification	$\exists x. t$	$\exists x. t$	<i>for some x : t</i>
$\epsilon$ -term	$\epsilon x. t$	$\epsilon x. t$	<i>an x such that: t</i>
Conditional	$\text{if } t \text{ then } t_1 \text{ else } t_2$	$(t \rightarrow t_1, t_2)$	<i>if t then t<sub>1</sub> else t<sub>2</sub></i>

Table 6.1: HOL Notation for Higher Order Logic Terms

3. Enter the HOL equivalent of  $\forall x y. P(x) \supset Q(y)$ , without explicitly specifying types. What do you get and why?
4. Enter the HOL equivalent of  $\exists (x : \text{num}). R(x : \alpha)$ . What happens and why?
5. Enter the HOL equivalent of  $\neg \forall x. P(x) \vee Q(x) = \exists x. \neg P(x) \wedge \neg Q(x)$
6. Enter the HOL equivalent of the English sentence, All people are mortal, where  $P(x)$  represents x is a person and  $M(x)$  represents x is mortal.
7. Enter the HOL equivalent of the English sentence, Some people are funny, where  $\text{Funny}(x)$  denotes x is funny.

# Constructing and Deconstructing HOL Terms

---

One of the benefits of using systems such as HOL is the ability to extend the syntax and semantics of HOL *safely*, i.e., in ways that guarantee logical soundness. This includes the capability to introduce new syntax and semantics, such as the syntax of our access-control logic and its associated Kripke semantics. We can also introduce the inference rules of the access-control logic as sound HOL inference rules.

## 7.1 Introduction to Constructing and Deconstructing HOL Terms

As we will see in the next lab, inference rules in HOL are ML functions that return HOL theorems as results. Central to inference rules is the capability to construct and deconstruct HOL terms using built-in and customized ML functions. In this lab, we learn how this is done.

Table 7.1 shows the predicates (tests), deconstructor, and constructor functions for common HOL formulas. When interacting directly with HOL through the ML interpreter, we can input directly the HOL terms we want. However, when we build specialized inference rules, we need to be able to deconstruct HOL terms and reassemble them in useful ways.

### Example 7.1

Our first example shows how to deconstruct the simplest of HOL terms, a variable  $x$  with HOL type  $\alpha$ . In the following session, observe:

1. `dest_var` deconstructs a variable ```x:'a``` into its component parts: its name given by the **string** `"x"`, and its HOL type, ```:'a```. Notice that the value returned by `dest_var` is an ML pair (Cartesian product) `string * hol_type`.
2. We can reconstruct the variable ```x:'a``` by using `mk_var` and applying it to the pair `"x", ``:'a```.

To reproduce the following session, start a HOL session within Emacs, make sure you click on **HOL**, **Printing switches**, **Show types**, and **Unicode**, to explicitly show types of terms and restrict HOL to the ASCII character set by toggling off Unicode.

ML Functions on HOL Terms				
Kind of term	HOL notation	Predicate	Deconstructor	Constructor
Variable	$x$	is_var	dest_var	mk_var
HOL Type	$:ty$	is_vartype	dest_vartype	mk_vartype
Negation	$\sim t$	is_neg	dest_neg	mk_neg
Disjunction	$t_1 \vee t_2$	is_disj	dest_disj	mk_disj
Conjunction	$t_1 \wedge t_2$	is_conj	dest_conj	mk_conj
Implication	$t_1 \Rightarrow t_2$	is_imp	dest_imp	mk_imp
Equality	$t_1 = t_2$	is_eq	dest_eq	mk_eq
$\forall$ -quantification	$\lambda x. t$	is_forall	dest_forall	mk_forall
$\exists$ -quantification	$\exists x. t$	is_exists	dest_exists	mk_exists
$\epsilon$ -term	$@x. t$	is_select	dest_select	mk_select
Conditional	if $t$ then $t_1$ else $t_2$	is_cond	dest_cond	mk_cond
Function application	$t_1 \ t_2$	is_comb	dest_comb	mk_comb
Lambda abstraction	$\lambda x. f$	is_abs	dest_abs	mk_abs

Table 7.1: Predicates, Deconstructors, and Constructors for HOL Terms

```

- (* A simple variable of type 'a *)
  ``x:'a``;
> val it = ``(x : 'a)`` : term
- (* Test that it's a variable *)
- is_var ``x:'a``;
> val it = true : bool
- (* Deconstruct it into its component parts: its name and type *)
dest_var ``x:'a``;
> val it = ("x", ``:'a``) : string * hol_type
- (*****
(* Notice that the value returned is a pair consisting of a *)
(* string and hol_type. We can reconstruct the variable by *)
(* giving the same value to mk_var. *)
(*****
val (varString, holType) = dest_var ``x``;
mk_var(varString, holType);
<<HOL message: inventing new type variable names: 'a>>
> val varString = "x" : string
  val holType = ``:'a`` : hol_type
- > val it = ``(x : 'a)`` : term

```

1



**Remember HOL is higher order:** Remember that HOL is higher order, i.e., functions are values to other functions and are returned as values. This means that *variables* are functions, too. Consider the following session where we deconstruct and reconstruct a variable that is a function.

```

- val (funName, funType) = dest_var ``f:'a -> 'b``;
> val funName = "f" : string
  val funType =
    ``:'a -> 'b`` : hol_type
- mk_var(funName, funType);
> val it =
  ``(f : 'a -> 'b)``
  : term

```

2

## 7.2 Manipulating HOL terms

The capability to take terms apart and put them together again in different ways using ML functions, is central to the inner workings of HOL. In Table 7.1, the following naming convention is used:

- Predicates testing to see if a term follows a pattern (e.g., is a variable, conjunction, disjunction, implication, etc.) start with `is_`.
- Destructors that decompose a term into its immediate constituent parts start with `dest_`.
- Constructors that create terms from its immediate constituent parts start with `mk_`.

In the examples that follow, we show how to take HOL terms apart and put them together in various ways. We start with a specific example, then show how to generalize the process by writing special-purpose functions in ML.

### Example 7.2

Suppose we want to change the variables in a function application, such as

`((f : 'a -> 'b) (x : 'a))`, to `((f : 'a -> 'b) (y : 'a))`, and do so by writing a function that takes the function application, a new variable name, and returns the function applied to the new variable.

A good way to start is to develop the process step-by-step in ML with a specific input values in such a way that it is easy to generalize into an ML function. The following session starts with specific values assigned to variable names `combTerm` and `newTerm2Name`. These two names will be the parameters of the final function we create for the general case.

```
- val combTerm = `(f x)`;
<<HOL message: inventing new type variable names: 'a, 'b>>
> val combTerm =
  `(f : 'a -> 'b) (x : 'a)`
  : term
- val newTerm2Name = "y";
> val newTerm2Name = "y" : string
```

3

Continuing on, the next session shows the use of what will become *internal variables* `term1`, `term2`, and `term2Type`. These internal variables are assigned values that are the components of `combTerm`.

```
- val (term1, term2) = dest_comb combTerm;
> val term1 =
  `(f : 'a -> 'b)`
  : term
  val term2 = `(x : 'a)` : term
- val (_, term2Type) = dest_var term2;
> val term2Type = `:'a` : hol_type
```

4

We now have all the components we need to assemble the output. First, we make the new term with `mk_var` applied to `(newTerm2Name, term2Type)`. Second, we create the function application using `mk_comb`.

```
- val newTerm2 = mk_var(newTerm2Name, term2Type);
> val newTerm2 = `(y : 'a)` : term
- mk_comb(term1, newTerm2);
> val it =
  `(f : 'a -> 'b) (y : 'a)`
  : term
```

5

Finally, we create the function `exampleFunction combTerm newTerm2Name` and use the above within an ML `let` expression, where the internal variables are surrounded by `let` and `in`, followed by the final step `mk_comb (term1, newTerm2)`, and concluded by `end`.

```

- fun exampleFunction combTerm newTerm2Name =
let
  val (term1, term2) = dest_comb combTerm
  val (_, term2Type) = dest_var term2
  val newTerm2 = mk_var(newTerm2Name, term2Type)
in
  mk_comb(term1, newTerm2)
end;

> val exampleFunction = fn : term -> string -> term
- exampleFunction ``foo bar`` "hoo";
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it =
  `` (foo : 'a -> 'b) (hoo : 'a) ``
   : term

```

◇

### Example 7.3

Suppose we wish to take HOL terms of the form  $t_1 \wedge t_2$  and reassemble them to form the term  $t_2 \wedge t_1$ . The steps we take are the following:

1. deconstruct  $t_1 \wedge t_2$  using `dest_conj` to get the individual components  $t_1$  and  $t_2$ , then
2. construct the term  $t_2 \wedge t_1$  using  $t_1$  and  $t_2$  with `mk_conj`.

As an illustration, suppose we have the HOL term ```p /\ q```. We use the steps above and follow the development process used in the previous example, i.e., naming input parameters, decomposing them into their parts named by internal variables, and finally reassembled into the desired output.

```

- val conjTerm = ``p /\ q``;
> val conjTerm =
  `` (p : bool) /\ (q : bool) ``
   : term
- val (t1, t2) = dest_conj conjTerm;
> val t1 = `` (p : bool) `` : term
  val t2 = `` (q : bool) `` : term
- mk_conj (t2, t1);
> val it = `` (q : bool) /\ (p : bool) `` : term

```

Following the identical approach as the previous example, we put the above into a function using a `let` expression.

```

- fun swapConj conjTerm =
let
  val (t1, t2) = dest_conj conjTerm
in
  mk_conj (t2, t1)
end;
> val swapConj = fn : term -> term
- swapConj `` (!x:'a. P x) /\ (?y:'b. Q y) ``;
> val it =
  `` (? (y : 'b). (Q : 'b -> bool) y) /\
    ! (x : 'a). (P : 'a -> bool) x ``
   : term

```

◇



**Example 7.4**

Suppose we wish to create a function `function4 forallTerm` that takes HOL terms of the form  $\forall x.P(x)$  and returns terms of the form  $\forall P.(\forall x.P(x) = \exists x.\neg P(x))$ , which is *not generally true*. We are just considering how to create the function. Following our developmental process, we give the input value a name, store the component values as local variables, and assemble the final output from the parts.

9

```

- val forallTerm = ``!x.P x``;
<<HOL message: inventing new type variable names: 'a>>
> val forallTerm =
  ``!(x : 'a). (P : 'a -> bool) x``
  : term
- val (var,scope) = dest_forall forallTerm;
> val var = ``(x : 'a)`` : term
  val scope =
    ``(P : 'a -> bool) (x : 'a)``
    : term
- val (term1,term2) = dest_comb scope;
> val term1 =
  ``(P : 'a -> bool)``
  : term
  val term2 = ``(x : 'a)`` : term
- val scope2 = mk_neg scope;
> val scope2 =
  ``~(P : 'a -> bool) (x : 'a)``
  : term
- val existsTerm = mk_exists(var,scope2);
> val existsTerm =
  ``?(x : 'a). ~(P : 'a -> bool) x``
  : term
- val eqTerm = mk_eq(forallTerm,existsTerm);
> val eqTerm =
  ``(! (x : 'a). (P : 'a -> bool) x) <=> ?(x : 'a). ~P x``
  : term
- mk_forall(term1,eqTerm);
> val it =
  ``!(P : 'a -> bool). (! (x : 'a). P x) <=> ?(x : 'a). ~P x``
  : term

```

Putting the appropriate parts within a `let` expression, we get the following.

10

```

- fun function4 forallTerm =
let
  val (var,scope) = dest_forall forallTerm
  val (term1,term2) = dest_comb scope
  val scope2 = mk_neg scope
  val existsTerm = mk_exists(var,scope2)
  val eqTerm = mk_eq(forallTerm,existsTerm)
in
  mk_forall(term1,eqTerm)
end;
> val function4 = fn : term -> term
- function4 ``!y.Q y``;
<<HOL message: inventing new type variable names: 'a>>
> val it =
  ``!(Q : 'a -> bool). (! (y : 'a). Q y) <=> ?(y : 'a). ~Q y``
  : term

```



## 7.3 Exercises

In the following problems, enable HOL's *Show types* capability and disable *Unicode* so only ASCII characters are displayed.

**Exercise 7.3.1** Create a function `andImp2Imp` term that operates on terms of the form  $p \wedge q \supset r$  and returns  $p \supset q \supset r$ .

```
- andImp2Imp `` (p /\ q) ==> r ``;
> val it =
  `` (p : bool) ==> (q : bool) ==> (r : bool) ``
  : term
```

11

**Exercise 7.3.2** Create a function `impImpAnd` term that operates on terms of the form  $p \supset q \supset r$  and returns  $p \wedge q \supset r$ . Show that `impImpAnd` reverses the effects of `andImp2Imp`, and vice versa.

```
- impImpAnd `` p ==> q ==> r ``;
> val it =
  `` (p : bool) /\ (q : bool) ==> (r : bool) ``
  : term
- impImpAnd (andImp2Imp `` (p /\ q) ==> r ``);
> val it =
  `` (p : bool) /\ (q : bool) ==> (r : bool) ``
  : term
- andImp2Imp (impImpAnd `` p ==> q ==> r ``);
> val it =
  `` (p : bool) ==> (q : bool) ==> (r : bool) ``
  : term
```

12

**Exercise 7.3.3** Create a function `notExists` term that operates on terms of the form  $\neg \exists x. P(x)$  and returns  $\forall x. \neg P(x)$ .

```
- notExists `` ~?z.Q z ``;
<<HOL message: inventing new type variable names: 'a>>
> val it =
  `` ! (z : 'a). ~ (Q : 'a -> bool) z ``
  : term
```

13

# Introduction to Forward Proofs in HOL

Our objectives for this first lab doing proofs in HOL include:

- Relating the structure of HOL theorems to “pencil and paper” theorems you likely have seen in symbolic logic courses.
- Relating proofs you have seen in symbolic logic courses to proofs using HOL’s inference rules.

Since 1988, when the first stable version of HOL (HOL88) was introduced, the *safe type* of HOL is the ML type `:thm`. *Safe type* means that everything in ML of type `:thm` is a **theorem** in logic. The implications of making the statement is that **unsound** statements are not provable in HOL. That is, it is not possible, using the inference rules of HOL, to go wrong and prove something that is not a theorem.

The pragmatic and practical implications are many. For engineers HOL (and proof checkers like HOL) provide the highest degree of assurance of correctness, repeatability, and reuse. Theorems proved in HOL are truly theorems. The ML functions that are used to prove the theorems of interest are completely sound and open to inspection.

HOL’s extensive collection of proved theories gives engineers, whose focus is assurance of correctness and security, the capability to soundly reuse and build upon a solid foundation created by others.

## 8.1 Structure of HOL Theorems

In the following examples, we have turned off Unicode so that HOL prints its terms using only the ASCII character set. We also explicitly show the assumptions in theorems by clicking on **HOL**, **Printing switches**, and **Show assumptions**. When helpful, we will also show types explicitly.

### Example 8.1

Our first theorem in HOL is `TRUTH`, shown below.

```
- TRUTH;  
> val it = [] |- T : thm
```

1

What we observe is the following:

- The value of `TRUTH` in ML is `[] |- T : thm`, i.e., something we will recognize as a theorem in HOL with an ML type of `:thm`.
- HOL theorems are denoted by the `|-` symbol, pronounced “*turnstile*”, which is usually typeset in logic as  $\vdash$ .
- To the left of `|-` is a list of boolean terms in HOL—these are the **hypotheses** or **assumptions**. In the case of `TRUTH`, there are no hypotheses, hence the list is empty.

- To the right of `| -` is a single boolean term in HOL—this is the **conclusion**.

What makes a theorem a theorem in HOL is this: for any assignment of values to atomic terms in the assumptions, whenever *all the assumptions are true, the conclusion is also true*.  $\vdash$  **denotes theoremhood**.

The structure  $\{\text{boolean terms}\} \vdash \text{boolean term}$  is called a **sequent** in symbolic logic.

**HOL theorems are sequents of the form  $\Gamma \vdash t$**

In sequents  $\Gamma \vdash t$ ,  $\Gamma$  is a **set** of boolean terms, i.e., the hypotheses, and  $t$  is a boolean term that is the conclusion. **Note: in HOL,  $\Gamma$  is represented by a list of boolean terms, whereas in the documentation of inference rules  $\Gamma$  is a set.** There is no inconsistency here, but it is essential that your proofs not rely on any particular ordering of hypotheses in the assumption list  $\Gamma$ .

The fact that HOL theorems and HOL terms are of different types is important. Terms are *components* of HOL theorems. This is part of why `:thm` is the safe type in HOL.

To illustrate how HOL terms are components of sequents, look at the following example. ◇

### Example 8.2

As all HOL theorems are structured as sequents, theorems are decomposable into their constituent parts using the ML accessor functions `hyp` and `concl`. Their ML type signatures are shown below.

```
- hyp;
> val it = fn : thm -> term list
- concl;
> val it = fn : thm -> term
```

2

We use `hyp` and `concl` to take apart theorems and when necessary, manipulate their constituent terms. Applying these accessor functions to `TRUTH` yields the following.

```
- hyp TRUTH;
> val it = [] : term list
- concl TRUTH;
> val it = ``T`` : term
```

3

The list of hypotheses in `TRUTH` is empty. The conclusion of `TRUTH` is `T`, i.e. *true*.

At this point, you may be wondering if there is a corresponding *constructor* function for type `:thm` in ML. The answer is yes, but we **never use it**. The whole idea is to **introduce theorems using the inference rules of HOL**.

There are simple ways to print out HOL theories to detect “cheats”. Unproved theorems (and theorems imported from systems outside HOL) have a different “color” or tag. Semanticists sometimes do “what if” explorations to see what would happen if certain statements were theorems. As our primary purpose is assurance of correctness and security we will not be introducing unproved theorems, ever. ◇

## 8.2 Inference Rules in HOL

Figure 8.1 at the end of this lab is a (very small) subset of the built-in inference rules in HOL. HOL comes with many inference rules already implemented (see the 1156 pages of the HOL Reference Manual to verify this). What will become apparent soon is that while these built-in rules are useful, perhaps as great a capability is for us to *create sound inference rules of our own specialized to our particular application*.

### Example 8.3

Our first inference rule in HOL is [ASSUME](#). If you go to the HOL reference page (double click on </usr/local/share/HOL/help/HOLindex.html> to bring up the reference page, then go to the bottom of the page, click on [IDENTIFIERS](#), and look up [ASSUME](#)), you will find the following description.

$$\text{ASSUME } t \quad \frac{}{\{t\} \vdash t} \quad (8.1)$$

The above says that the theorem returned by [ASSUME](#) when applied to a boolean term `t:bool` is the theorem  $\{t\} \vdash t$ . The HOL session creating the theorem and deconstructing it to its components, i.e., its hypotheses and conclusion, is shown below.

```
- val th1 = ASSUME ``t:bool``;
> val th1 =
  [(t :bool)] |- (t :bool)
  : thm
- val asList = hyp th1;
> val asList = [``(t :bool)``] :
  term list
- val concl1 = concl th1;
> val concl1 = `` (t :bool) `` :
  term
```

4

To see why the above is sound, recall that if the conclusion is guaranteed to be true whenever all of the assumptions are true, then the assumptions and conclusion combined are a theorem in the form of a sequent. As `t` is both the conclusion and the only hypothesis, whenever `t` is true as a hypothesis, then `t` is true as a conclusion. Thus,  $\{t\} \vdash t$  is a theorem.  $\diamond$

### Example 8.4

The [DISCH](#) rule is described by:

$$\text{DISCH } u \quad \frac{\Gamma \vdash t}{\Gamma - \{u\} \vdash u \supset t} \quad (8.2)$$

In this rule, a boolean term `u` in the set of hypotheses  $\Gamma$  is deleted from  $\Gamma$  and introduced into the conclusion as an *implication*  $u \supset t$ , where  $\supset$  denotes logical implication.

To see how this works, we introduce  $\{p\} \vdash p$  as a theorem using [ASSUME](#) and then use the [DISCH](#) `p` inference rule to derive  $\{\} \vdash p \supset p$ .

```
- val th1 = ASSUME ``p:bool``;
> val th1 = [p] |- p : thm
- val th2 = DISCH ``p:bool`` th1;
> val th2 = [] |- p ==> p : thm
```

5

`th2` is a theorem that has no assumptions and states that  $p \supset p$  is *always true for all truth values assigned to  $p$* . In other words,  $p \supset p$  is a *tautology*. This fact is easily seen if you recall that  $p \supset p$  is equivalent to  $\neg p \vee p$  using the definition of implies.  $\diamond$

### Example 8.5

Once we have done a proof, it's a good idea to tidy up after ourselves. By that we mean to package everything up into a single function that keeps all the inferences inside the function as local operations. This we have done before in ML (and Haskell) using `let` expressions.

```
val example5Thm =
  let
    val th1 = ASSUME ``p:bool``
  in
    DISCH(hd(hyp th1)) th1
  end;
> val example5Thm =
    [] |- p ==> p : thm
```

6

The above session shows how we use functional programming to our advantage (and gives a glimpse of how we will create our own inference rules). Notice that `val th1 = ASSUME ``p:bool``` appears within the scope of a `let` expression. This means that `th1` is a local value. Notice too, that the argument of `DISCH` is `hd(hyp th1)`. We could have used ```p:bool``` instead and gotten the same result. Nevertheless, we did it this way to show that we can take advantage of HOL's destructor functions to access the components of theorems. When individual terms become large and unwieldy, it is much easier to use accessor functions than the actual terms.  $\diamond$

## 8.3 Relating Proofs in HOL to Traditional Proofs

In this section we show the relationship between proofs in HOL and the traditional pencil and paper symbolic logic proofs you did in the past.

### Example 8.6

Consider the following rules of inference that are typically used in symbolic logic.

$$\begin{array}{lll}
 \text{Modus Ponens} & \frac{p \supset q \quad p}{q} & \text{Simplification} \quad \frac{p \wedge q}{p} \\
 & & \text{Conjunction} \quad \frac{p \quad q}{p \wedge q} \\
 \text{Addition} & \frac{p}{p \vee q} &
 \end{array}$$

Suppose our goal is to prove:

1.  $A \wedge B$
2.  $(A \vee C) \supset D \therefore A \wedge D$ ,

that is, under the assumptions  $A \wedge B$  and  $(A \vee C) \supset D$ , we can derive  $A \wedge D$ . Our pencil and paper proof using the above inference rules might look like:

**Figure 8.1** HOL Inference Rules

$ASSUME\ t \quad \frac{}{\{t\} \vdash t}$	$DISCH\ u \quad \frac{\Gamma \vdash t}{\Gamma - \{u\} \vdash u \supset t}$	$REFL\ t \quad \frac{}{\vdash t = t}$
$CONJ \quad \frac{\Gamma_1 \vdash t_1 \quad \Gamma_2 \vdash t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2}$	$CONJUNCT1 \quad \frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_1}$	$CONJUNCT2 \quad \frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_2}$
$MP \quad \frac{\Gamma_1 \vdash t_1 \supset t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$	$EQ\_MP \quad \frac{\Gamma_1 \vdash t_1 = t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$	
$IMP\_ANTISYM\_RULE \quad \frac{\Gamma_1 \vdash t_1 \supset t_2 \quad \Gamma_2 \vdash t_2 \supset t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \Leftrightarrow t_2}$		
$IMP\_TRANS \quad \frac{\Gamma_1 \vdash t_1 \supset t_2 \quad \Gamma_2 \vdash t_2 \supset t_3}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \supset t_3}$	$NOT\_ELIM \quad \frac{\Gamma \vdash \neg t}{\Gamma \vdash t \supset F}$	$NOT\_INTRO \quad \frac{\Gamma \vdash t \supset F}{\Gamma \vdash \neg t}$
$DISJ1 \quad \frac{\Gamma \vdash t_1}{\Gamma \vdash t_1 \vee t_2}$	$DISJ2 \quad \frac{\Gamma \vdash t_1}{\Gamma \vdash t_2 \vee t_1}$	$IMP\_ELIM \quad \frac{\Gamma \vdash s \supset t}{\Gamma \vdash \neg s \vee t}$
$DISJ\_IMP \quad \frac{\Gamma \vdash t_1 \vee t_2}{\Gamma \vdash \neg t_1 \supset t_2}$		
$GEN\ x\ where\ x\ not\ free\ in\ \Gamma \quad \frac{\Gamma \vdash t}{\Gamma \vdash \forall x. t}$	$GENL\ [x_1, \dots, x_n]\ where\ no\ x_i\ is\ free\ in\ \Gamma \quad \frac{\Gamma \vdash t}{\Gamma \vdash \forall x_1 \dots x_n. t}$	
$ISPEC\ t : ty' \ where\ t\ is\ free\ for\ x\ in\ t_m, \text{ and } ty' \text{ is an instance of } ty \quad \frac{\Gamma \vdash \forall (x : ty). t_m}{\Gamma \vdash t_m[t/x]}$		
$ISPECL\ [t_1, \dots, t_n]\ where\ t_i\ is\ free\ for\ x_i\ in\ t_m \quad \frac{\Gamma \vdash \forall x_1 \dots x_n. t}{\Gamma \vdash t[t_1, \dots, t_n/x_1, \dots, x_n]}$		
$SUBS\ [\Gamma_1 \vdash t_1 = v_1; \dots; \Gamma_n \vdash t_n = v_n] (\Gamma \vdash t) \quad \frac{\Gamma_1 \vdash t_1 = v_1 \dots \Gamma_n \vdash t_n = v_n \quad \Gamma \vdash t}{\Gamma_1 \cup \dots \cup \Gamma_n \cup \Gamma \vdash t[v_1, \dots, v_n/t_1, \dots, t_n]}$		
$REWRITE\_RULE\ [\Gamma_1 \vdash t_1; \dots; \Gamma_n \vdash t_n] \quad \frac{\Gamma \vdash t}{\Gamma_1 \cup \dots \cup \Gamma_n \cup \Gamma \vdash t'}$	Uses tautologies in the ML list <code>basic_rewrites</code> as well as the list of theorems supplied by the user. There is no specified ordering to the rewrites. It may not terminate.	

- |                           |                    |
|---------------------------|--------------------|
| 1. $A \wedge B$           | assumption         |
| 2. $(A \vee C) \supset D$ | assumption         |
| 3. $A$                    | 1, Simplification  |
| 4. $A \vee C$             | 3, Addition        |
| 5. $D$                    | 2, 4, Modus Ponens |
| 6. $A \wedge D$           | 3, 5 Conjunction   |

In the above proof, each of the steps is a formula corresponding to a HOL term introduced as an assumption or by an inference rule of the logic.

Recall, in HOL our inference rules return *theorems as sequents*. We recast the above proof using HOL's sequents and inference rules, as shown in Figure 8.1.

- |   |                                 |
|---|---------------------------------|
| 1. $\{A \wedge B\} \vdash A \wedge B$   | ASSUME                          |
| 2. $\{(A \vee C) \supset D\} \vdash (A \vee C) \supset D$                         | ASSUME                          |
| 3. $\{A \wedge B\} \vdash A$  | 1, CONJUNCT1                    |
| 4. $\{A \wedge B\} \vdash A \vee C$   | 3, DISJ1 C:bool                 |
| 5. $\{(A \vee C) \supset D, A \wedge B\} \vdash D$                                | 2, 4 MP                         |
| 6. $\{(A \vee C) \supset D, A \wedge B\} \vdash A \wedge D$                       | 3, 5 CONJ                       |
| 7. $\{A \wedge B\} \vdash ((A \vee C) \supset D) \supset (A \wedge D)$            | 6, DISCH $(A \vee C) \supset D$ |
| 8. $\{\} \vdash (A \wedge B) \supset ((A \vee C) \supset D) \supset (A \wedge D)$ | 7, DISCH $A \wedge B$           |

The following session shows the interactive development of the proof.

```

- val th1 = ASSUME ``A /\ B``;
> val th1 =
  [A /\ B] |- A /\ B
  : thm
- val th2 = ASSUME ``A \/ C ==> D``;
> val th2 =
  [A \/ C ==> D] |- A \/ C ==> D
  : thm
- val th3 = CONJUNCT1 th1;
> val th3 =
  [A /\ B] |- A
  : thm
- val th4 = DISJ1 th3 ``C:bool``;
> val th4 =
  [A /\ B] |- A \/ C
  : thm
- val th5 = MP th2 th4;
> val th5 =
  [A /\ B, A \/ C ==> D] |- D
  : thm
- val th6 = DISCH (hd(hyp(th2))) th5;
> val th6 =
  [A /\ B] |- (A \/ C ==> D) ==> D
  : thm
- val th7 = DISCH (hd(hyp(th1))) th6;
> val th7 =
  [] |- A /\ B ==> (A \/ C ==> D) ==> D
  : thm

```

7

Once we have developed the proof, we can package it up neatly with a `let` expression as shown below.

```

val example6Thm =
let
  val th1 = ASSUME ``A /\ B``
  val th2 = ASSUME ``A \/ C ==> D``
  val th3 = CONJUNCT1 th1
  val th4 = DISJ1 th3 ``C:bool``
  val th5 = MP th2 th4
  val th6 = DISCH (hd(hyp(th2))) th5
in
  DISCH (hd(hyp(th1))) th6
end;
> val example6Thm =
  [] |- A /\ B ==> (A \/ C ==> D) ==> D
  : thm

```

8

◇

### Example 8.7

In Example 6, we proved the theorem

$$\{\} \vdash (A \wedge B) \supset ((A \vee C) \supset D) \supset D. \quad (8.3)$$

We can *universally quantify* the four variables because none of them appear free in the assumptions of the sequent. Doing so states that the relation holds for all values of  $A$ ,  $B$ ,  $C$ , and  $D$ .

$$\{\} \vdash \forall A B C D. (A \wedge B) \supset ((A \vee C) \supset D) \supset D. \quad (8.4)$$



The HOL inference rules that introduce universal quantifiers are [GEN](#) and [GENL](#). They are described in Figure 8.1. The session below uses

```
GENL [``A:bool``, ``B:bool``, ``C:bool``, ``D:bool``].
```

```
- val example7Thm =
  GENL [``A:bool``, ``B:bool``, ``C:bool``, ``D:bool``] example6Thm;
> val example7Thm =
  [] |- !A B C D. A /\ B ==> (A \/ C ==> D) ==> D
  : thm
```

9

◇

## 8.4 Exercises

In the following problems, enable HOL's *Show types* capability and disable *Unicode* so only ACSII characters are displayed. In all your problems, **package up your proofs into a single function that returns the theorem.**

**Exercise 8.4.1** *Prove the following theorem.*

```
> val problem1Thm =
  [] |- p ==> (p ==> q) ==> (q ==> r) ==> r
  : thm
```

10

**Exercise 8.4.2** *Prove the following theorem:*

```
> val conjSymThm =
  [] |- p /\ q <=> q /\ p
  : thm
```

11

**Hint:** first prove the following two theorems:  $\{\} \vdash p \wedge q \supset q \wedge p$  and  $\{\} \vdash q \wedge p \supset p \wedge q$  and then use the [IMP\\_ANTISYM\\_RULE](#).

**Exercise 8.4.3** *Extend your proof in Problem 2 by one step and prove:*

```
> val conjSymThmAll =
  [] |- !p q. p /\ q <=> q /\ p
  : thm
```

12

BLANK PAGE

# Goal-Oriented Proofs

---

At this point, we now know how to use Haskell and ML as functional programming languages. So far, we have focused on using ML and ML functions to manipulate HOL terms and theorems. These proofs are known as *forward* style proofs. They start with sequents and end with sequents. The steps taken are typically small and sometimes intricate.

Now, we introduce *goal-oriented* proofs. These proofs start with the desired end as a goal and work *backwards towards* known theorems. If all paths and cases end in known theorems, then the reverse process (which is automated in HOL) produces a proof of the desired goal.

Goal-oriented proofs allow us to take bigger steps, and thus make proofs of larger goals more convenient. We still need our forward proof skills as often we will wish to manipulate our assumptions that will take the form of theorems. We will devote an entire lab to how this is done.

Once we have the basics down in terms of goal-oriented proofs, we will introduce algebraic models of cryptographic operations. By so doing, we will be able to use cryptographic operations as system components and reason about the results. Proving theorems about cryptographic operations is easiest done using goal-oriented proofs.

## 9.1 Goal-Oriented Proofs

Most proofs in HOL are done starting with a *goal* to be proved as opposed to starting with a theorem, as we did for the previous assignment. Goal-oriented proofs (otherwise known as *backward* proofs) start with a proposition we wish to prove. The proof proceeds by simplifying the goal to simpler subgoals until they all correspond to known theorems. The functions that are applied to goals to produce simpler goals are known as *tactics*. In this homework, you will become familiar with some basic tactics and their operation, as well as write your own tactics in ML.

### 9.1.1 Basic Techniques

As a general rule, do all of your HOL work by starting a HOL session within Emacs. Typically, you will have an sml file within which you will write your definitions and proofs. After opening your sml file, start a HOL process by clicking **HOL**, **Process**, and either **Start HOL horizontal** or **Start HOL vertical**. As you develop your HOL definitions and proofs, write them down first in your sml file, then copy and paste your latest developments into the HOL window.

**Setting Goals** Goals have the same structure as sequents. Goals have a list of assumptions  $\Gamma$  and a conclusion  $t$ . They are represented as  $\Gamma \vdash t$ . While goals have the same structure as sequents, **they are not theorems** in HOL. Recall that theorems  $\Gamma \vdash t$  are the *safe type* in ML, i.e., all ML objects of type `thm` are guaranteed to be mathematically valid. A goal-oriented proof is done when a goal corresponds to an existing theorem.

Goals are set using the ML function `set_goal`. The ML type of `set_goal` is shown below in Session 1.

```
- set_goal;
> val it = fn : term list * term -> proofs
```

1

Notice that `set_goal` takes a pair (`term list`  $\times$  `term`) and returns `proofs`. The input pair is a goal of the form  $\Gamma \vdash t$ .

Suppose we wish to prove the goal  $\{A, B\} \vdash A \wedge B$ . We apply `set_goal` to the pair (`[“A : bool”, “B : bool”, “A  $\wedge$  B”]`) and the result is a proof object as shown below. The result is shown below in Session 2. The proof manager reports that there is one proof in the current HOL session. The first proof has an initial goal of  $A \wedge B$  under the two assumptions in the assumption list: assumption 0 being  $B$  and assumption 1 being  $A$ .

```
- set_goal(["A:bool", "B:bool", "A /\ B"]);
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      A /\ B
      -----
      0.  B
      1.  A

  : proofs
```

2

We quickly surmise that the above is true because whenever both  $A$  and  $B$  are true in the assumption list, then  $A \wedge B$  must be true in the assumptions. The task now becomes how to convince HOL of this. This leads us directly to tactics and tacticals—the functions used in developing HOL proofs.

### 9.1.2 Tactics

In the same way that goals in backwards proofs correspond to theorems in forward proofs, *tactics* in backwards proofs correspond to inference rules in forward proofs. Specifically, just as inference rules in forward proofs move us from theorem to theorem, tactics in backwards proofs move us from goal to goal.

As our first introduction to the use of tactics, consider `CONJ_TAC`. It is described by:

$$\frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_1 \quad \Gamma \vdash t_2} \text{ CONJ\_TAC}.$$

Its interpretation is as follows: if we have the goal  $\Gamma \vdash t_1 \wedge t_2$ , then we can prove it if we can prove both the (simpler) goals  $\Gamma \vdash t_1$  and  $\Gamma \vdash t_2$ . Similar to forward inference rules that state if you have the sequents above the (single) line, then you can write the sequent below the line, the rules for tactics are if you have the goal above the (double line) then you can reduce the goal to the subgoals below the line by applying the named tactic to the goal above the line.

Recall the description of `CONJ`.

$$\text{CONJ} \quad \frac{\Gamma_1 \vdash t_1 \quad \Gamma_2 \vdash t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2}.$$

It is no coincidence that `CONJ_TAC` looks like an upside-down version of `CONJ`. Tactics do two things:

1. Simplify goals into one or more subgoals.
2. Justify why proving the subgoals will prove the goals, i.e., what the corresponding forward inference rule should be applied to the sequents corresponding to the subgoals.

A simple example will clarify what is happening.

Our current proof goal is to show  $\{A, B\} \vdash A \wedge B$  is true. We know that this is a provable goal from the forward inference rule CONJ. In fact, the forward proof is straightforward, as shown below in HOL Session 3

```
- show_assums := true;
> val it = () : unit
- val th1 = ASSUME ``A:bool``;
> val th1 = [A] |- A : thm
- val th2 = ASSUME ``B:bool``;
> val th2 = [B] |- B : thm
- CONJ th1 th2;
> val it = [A, B] |- A /\ B : thm
```

3

How do we do the corresponding proof starting with the goal  $([``A:bool``, ``B:bool``, ``A /\ B``])$ ? Defining  $([``A``, ``B``, ``A /\ B``])$  as shown below in Session 4 we see as expected that the ML type of `goal1` is `term list * term`, i.e., the Cartesian product (pair) whose first element is a list of HOL terms, and whose second element is a HOL term.

```
- val goal1 = ([``A:bool``, ``B:bool``, ``A /\ B``]);
> val goal1 = ([``A``, ``B``, ``A /\ B``]) : term list * term
```

4

Looking at the ML type of the tactic `CONJ_TAC`, as shown below in Session 5, shows that `CONJ_TAC` takes as its argument a goal and returns a *pair* consisting of a *goal list* and a *function* whose input is a list of theorems and whose output is a theorem.

```
- CONJ_TAC;
> val it = fn : term list * term -> (term list * term) list * (thm list -> thm)
```

5

The proof of  $\{A, B\} \vdash A \wedge B$  using `CONJ_TAC` is as follows. First, we apply `CONJ_TAC` to `goal1` and get the corresponding justification function `just_fn`. Next, we apply `just_fn` to the theorems `th1` and `th2`. This is shown below in Session 6.

```
- CONJ_TAC goal1;
> val it = ([([``A``, ``B``, ``A /\ B``]), ([``A``, ``B``, ``B``]), fn) :
  (term list * term) list * (thm list -> thm)
- val (goal_list, just_fn) = it;
> val goal_list = ([([``A``, ``B``, ``A /\ B``]), ([``A``, ``B``, ``B``]) :
  (term list * term) list
  val just_fn = fn : thm list -> thm
- just_fn [th1, th2];
> val it = [A, B] |- A /\ B : thm
```

6

What is the magic within `CONJ_TAC`? There is none. It is simply the matter of defining the subgoals using the ML destructor function `dest_conj` on the input goal and defining the forward inference rule that is applied to the two theorems corresponding to the subgoals. The following is an equivalent function

Purpose	ML function	Abbreviation	Example
Setting top-level goal	set_goal	g	set_goal(['`A`', '`B`'], '`A /\ B`');
Applying a tactic to a proof	expand	e	e(CONJ_TAC);
Displaying the top-level goal	top_goal	none	top_goal();
Backing up one step in a proof	backup	b	b();

Table 9.1: Proof Management Functions

`Conj_tac` whose definition corresponds to the definition of the HOL built-in ML function `CONJ_TAC`. Recall, that lambda functions of the form  $\lambda x.t$  are defined in ML by `fn x => t`. Lambda functions are *anonymous* functions, i.e., functions with no names. The definition of `Conj_tac` is below in Session 7.

```

- fun Conj_tac (asl,term) =
let
  val (l,r) = dest_conj term
in
  ([ (asl,l), (asl,r)], fn [th1,th2] => CONJ th1 th2)
end;
! Toplevel input:
!   [ (asl,l), (asl,r)], fn [th1,th2] => CONJ th1 th2
!   ~~~~~
! Warning: pattern matching is not exhaustive
> val 'a Conj_tac = fn : 'a * term -> ('a * term) list * (thm list -> thm)

```

7

To demonstrate that `Conj_tac`—our version of `CONJ_TAC`—works, we repeat the previous backwards proof using `Conj_tac` as shown in Session 8. It produces the desired final theorem.

```

- Conj_tac goal1;
> val it = ([ ([ '`A`', '`B`'], '`A`' ), ([ '`A`', '`B`'], '`B`' ) ], fn) :
  (term list * term) list * (thm list -> thm)
- val (goal_list, just_fn) = it;
> val goal_list = [ ([ '`A`', '`B`'], '`A`' ), ([ '`A`', '`B`'], '`B`' ) ] :
  (term list * term) list
  val just_fn = fn : thm list -> thm
- just_fn [th1,th2];
> val it = [A, B] |- A /\ B : thm

```

8

## 9.2 Using Tactics and Tacticals

### 9.2.1 Tactics

Tactics are ML functions that operate on goals and return a pair consisting of a list of (sub)goals with a justification function that is a forward inference rule that proves a theorem corresponding to the goal when applied to a list of theorems corresponding to the subgoals. *Tacticals* are ML functions that take one or more tactics as arguments and returns a tactic. Keeping track of all the goals and justification functions manually is cumbersome.

Fortunately, HOL has a proof management library that manages the goals and justification functions automatically. We already used one of the proof management functions, `set_goal`, to set our top-level

goal  $\{A, B\} \vdash A \wedge B$ . To apply the tactic `CONJ_TAC` to the top-level goal in the proof, we apply the ML function `expand` to `CONJ_TAC`. The result is shown below in Session 9.

9

```

- set_goal(['A:bool', 'B:bool'], 'A /\ B');
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      A /\ B
      -----
    0. B
    1. A

    : proofs

- expand(CONJ_TAC);
OK..
2 subgoals:
> val it =

  B
  -----
    0. B
    1. A

  A
  -----
    0. B
    1. A

    : proof

```

We see from the above that the proof manager, as a result of applying `expand` to `CONJ_TAC` produced two subgoals:  $\{A, B\} \vdash A$  and  $\{A, B\} \vdash B$ . When using HOL interactively, we can only work on one goal at a time. That is, we can only work on the top goal of the goal stack. Which is the top goal on the goal stack? We use `top_goal` to find out. As Session 10 shows,  $\{A, B\} \vdash A$  is the top goal on the goal stack.

10

```

- top_goal();
> val it = (['A', 'B'], 'A') : term list * term

```

Recall, that we have  $\{A\} \vdash A$  and  $\{B\} \vdash B$  are theorems `th1` and `th2`. In a fashion corresponding to the forward proof of  $\{A, B\} \vdash A \wedge B$  we can use `th1` and `th2` to finish the proof using tactics. An ML function we can use is `ACCEPT_TAC` which we apply to `th1` to solve the first subgoal followed by `th2` to solve the last. The description of `ACCEPT_TAC` is

$$\frac{\Gamma \vdash t}{\text{ACCEPT\_TAC } \Gamma \vdash t}.$$

`ACCEPT_TAC` is a *theorem-tactic* as it takes as its first argument a `theorem` and returns a tactic. `ACCEPT_TAC` takes a theorem, in this case  $A \vdash A$  and when applied to the goal  $\{A, B\} \vdash A$  produces *no goals*. This is the backwards proof equivalent of a forward inference rule with no hypotheses.

Applying `ACCEPT_TAC th1` solves the top goal. The proof manager in HOL indicates that the goal is proved and prints out the remaining subgoal  $\{A, B\} \vdash B$ .

**Figure 9.1** Tactics

$\frac{\Gamma \text{ ?- } t_1 \wedge t_2}{\Gamma \text{ ?- } t_1 \quad \Gamma \text{ ?- } t_2}$	CONJ_TAC	$\frac{\Gamma \text{ ?- } \forall x.t}{\Gamma \text{ ?- } t[x'/x]}$	GEN_TAC	$\frac{\Gamma \text{ ?- } u \supset v}{\Gamma \cup \{u\} \text{ ?- } v}$	DISCH_TAC
$\frac{\Gamma \text{ ?- } t_1 = t_2}{\Gamma \text{ ?- } t_1 \supset t_2 \quad \Gamma \text{ ?- } t_2 \supset t_1}$	EQ_TAC	$\frac{\Gamma \text{ ?- } t}{\Gamma \text{ ?- } t}$	ACCEPT_TAC	$\Gamma \vdash t$	
$\frac{\Gamma \text{ ?- } t}{\Gamma \cup \{\text{terms due to resolution}\} \text{ ?- } t}$	RES_TAC				
$\frac{\Gamma \text{ ?- } t'}{\Gamma \text{ ?- } t}$	where $t'$ is $t$ rewritten using supplied theorems, assumptions, and built-in tautologies	ASM-REWRITE_TAC			
$\frac{\Gamma' \text{ ?- } t'}{\Gamma' \text{ ?- } t'}$	where $\Gamma'$ and $t'$ are obtained from $\Gamma \text{ ?- } t$ by eliminating the outermost connective	STRIP_TAC			

```

- val th1 = ASSUME ``A:bool``;
> val th1 = [A] |- A : thm
- expand(ACCEPT_TAC th1);
OK..

```

```

Goal proved.
[A] |- A

```

```

Remaining subgoals:
> val it =

```

```

    B
    -----
    0. B
    1. A
    : proof

```

11

We prove the remaining goal using theorem `th2` as shown below.

```

- val th2 = ASSUME ``B:bool``;
> val th2 = [.] |- B : thm
- expand(ACCEPT_TAC th2);
OK..

```

```

Goal proved.
[.] |- B

```

```

> val it =
    Initial goal proved.
    [...] |- A /\ B : proof

```

12

At this point, we have completed an *interactive goal-oriented proof* in much the same way as we did when doing forward inference proofs. Just as we did for forward proofs, we need to combine each step into a single function. To do this, we need to use *tacticals*, i.e., functions that take tactics as arguments and return tactics as results.

## 9.2.2 Tacticals

In this section, we introduce the *tacticals* (functions that take tactics as arguments and return a tactic as a result) `THENL` and `THEN`. These tacticals are often used to compose single proof functions from the application of individual tactics used to prove a goal interactively.



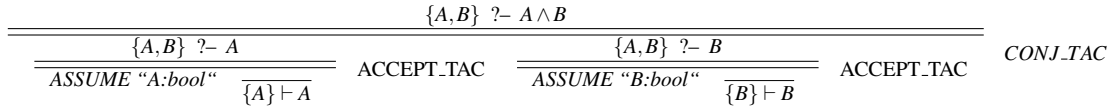
**Figure 9.2** Tactic Proof Tree

Figure 9.2 shows the proof tree of tactics used to prove the goal  $\{A, B\} \text{ ?- } A \wedge B$ . We note from Figure 9.2 that the first step was the application of `CONJ_TAC` to the top-level goal. This produced two subgoals:  $\{A, B\} \text{ ?- } A$  and  $\{A, B\} \text{ ?- } B$ . The first goal was proved using

```
ACCEPT_TAC (ASSUME ``A:bool``).
```

The second goal was proved using

```
ACCEPT_TAC (ASSUME ``B:bool``).
```

This type of proof structure occurs frequently, where a tactic is applied and yields several subgoals, to which each subgoal is operated on by a different tactic. In general, this situation is handled by the *infix tactical* `THENL`. In general, if tactic  $T_1$  produces  $n$  subgoals, each of which is operated on by tactics  $T_{21} \dots T_{2n}$ , we can package up the  $n+1$  tactics into a single tactic using `THENL`. Specifically,

$$T_1 \text{ THENL } [T_{21}, \dots, T_{2n}].$$

Session 13 shows the use of `THENL` to prove, in a single tactic, the goal  $\{A, B\} \text{ ?- } A \wedge B$

13

```

- show_assums := true;
> val it = () : unit
- set_goal([``A:bool``, ``B:bool``, ``A /\ B``]);
> val it =
  Proof manager status: 2 proofs.
  2. Completed goalstack: [A, B] |- A /\ B
  1. Incomplete goalstack:
    Initial goal:
    A /\ B
    -----
    0. B
    1. A

    : proofs
- expand(
  CONJ_TAC THENL
  [(ACCEPT_TAC (ASSUME ``A:bool``)),
   (ACCEPT_TAC (ASSUME ``B:bool``))]);
OK..
> val it =
  Initial goal proved.
  [A, B] |- A /\ B : proof

```

We can package the setting of a goal and its proof using `TAC_PROOF`. `TAC_PROOF` takes a pair consisting of a goal and a tactic proving the goal, and returns a theorem.

```
- val Theorem1 =
TAC_PROOF
(
  ([ ``A:bool``, ``B:bool``, ``A /\ B ``),
  (CONJ_TAC THENL
    [(ACCEPT_TAC (ASSUME ``A:bool``)),
     (ACCEPT_TAC (ASSUME ``B:bool``))])
);
> val Theorem1 = [A, B] |- A /\ B : thm
```

14

**An alternate and simpler proof** In HOL there are numerous tactics and inference rules. The tactics we have used thus far are very simple. A very powerful and useful tactic is `ASM_REWRITE_TAC`. This tactic rewrites the goal using built-in rewrites (e.g.,  $\sim\sim t = t$ ), a list of theorems supplied by the user, and the *set of assumptions in the goal*. The proof using `ASM_REWRITE_TAC` is shown below in Session 15.

15

```

- set_goal(['`A:bool`,`B:bool`,`A /\ B`']);
> val it =
  Proof manager status: 3 proofs.
  3. Completed goalstack: [A, B] |- A /\ B
  2. Completed goalstack: [A, B] |- A /\ B
  1. Incomplete goalstack:
      Initial goal:
      A /\ B
      -----
      0. B
      1. A

      : proofs
- expand(CONJ_TAC);
OK..
2 subgoals:
> val it =
  B
  -----
  0. B
  1. A

  A
  -----
  0. B
  1. A
  : proof
- expand(ASM_REWRITE_TAC[]);
OK..

Goal proved.
[A] |- A

Remaining subgoals:
> val it =
  B
  -----
  0. B
  1. A
  : proof
- expand(ASM_REWRITE_TAC[]);
OK..

Goal proved.
[B] |- B
> val it =
  Initial goal proved.
  [A, B] |- A /\ B : proof

```

We note that in this proof the same tactic `ASM_REWRITE_TAC []` is applied to both subgoals. While it is correct to write `CONJ_TAC THENL [ASM_REWRITE_TAC [], ASM_REWRITE_TAC []]`, it is more economical to use the `THEN` tactical. When we write  $T_1$  `THEN`  $T_2$ , what happens is tactic  $T_1$  is applied to a goal, then tactic  $T_2$  is applied to *all* subgoals. This is shown in Session 16.

16

```

- set_goal(['`A:bool`,`B:bool`,`A /\ B`]);
> val it =
  Proof manager status: 4 proofs.
  4. Completed goalstack: [A, B] |- A /\ B
  3. Completed goalstack: [A, B] |- A /\ B
  2. Completed goalstack: [A, B] |- A /\ B
  1. Incomplete goalstack:
      Initial goal:
      A /\ B
      -----
      0. B
      1. A

      : proofs
- expand(CONJ_TAC THEN
  ASM_REWRITE_TAC []);
OK..
> val it =
  Initial goal proved.
  [A, B] |- A /\ B : proof

```

**An even simpler proof** Rewriting tactics are generally quite powerful. As a final illustration with the current goal, we can prove the goal with a single application of `ASM_REWRITE_TAC` as shown below.

17

```

- set_goal(['`A:bool`,`B:bool`,`A /\ B`]);
> val it =
  Proof manager status: 5 proofs.
  5. Completed goalstack: [A, B] |- A /\ B
  4. Completed goalstack: [A, B] |- A /\ B
  3. Completed goalstack: [A, B] |- A /\ B
  2. Completed goalstack: [A, B] |- A /\ B
  1. Incomplete goalstack:
      Initial goal:
      A /\ B
      -----
      0. B
      1. A

      : proofs
- expand(ASM_REWRITE_TAC []);
OK..
> val it =
  Initial goal proved.
  [A, B] |- A /\ B : proof

```

### 9.2.3 Working with Assumptions

Consider the case where we have a goal  $\{p, p \supset q\} \vdash q$ . What we want to do is use Modus Ponens on the two assumptions to derive  $p$  and then prove the goal. The goal is shown below.

```

- set_goal(['`p:bool`', '`p==>q`'], '`q:bool`');
> val it =
  Proof manager status: 12 proofs.

  1. Incomplete goalstack:
    Initial goal:
    q
    -----
    0. p ==> q
    1. p

: proofs

```

18

If we try simple rewriting using the assumptions we make no progress.

```

- e(ASM_REWRITE_TAC []);
OK..
1 subgoal:
> val it =
  q
  -----
  0. p ==> q
  1. p

: proof

```

19

However, we can use the tactic `RES_TAC`, which does apply *resolution* (an operation that includes Modus Ponens) on the assumptions. This solves our goal.

```

- e(RES_TAC);
OK..

Goal proved.
[p, p ==> q] |- q
> val it =
  Initial goal proved.
  [p, p ==> q] |- q : proof

```

20

### 9.3 Using HOL within Emacs

Consult *Guide to HOL4 interaction and basic proofs*, which is part of the HOL documentation as a reference

Thus far, we have used a command line interface to HOL. By using the HOL interface in Emacs, we avoid copying portions of an sml file and pasting them into HOL. In the following example, open a new or an existing sml file within Emacs, start a HOL session, and turn off Unicode printing. Recall, we start a HOL session within Emacs by clicking **HOL** → **Process** → **Start HOL vertical (or horizontal) split**. Recall, we turn off (or on) Unicode printing by **HOL** → **Printing switches** → **Unicode**.

#### Example 9.1

Suppose we wish to prove the following:

$$\forall x y z. (x \Rightarrow (y \Rightarrow z)) = ((x \wedge y) \Rightarrow z)$$

Do the following to set up the goal.

## 1. Type the goal

```
!(x:bool) (y:bool) (z:bool). ((x /\ y) ==> z) = (x ==> (y ==> z))
```

into your sml file.

2. Highlight the goal using your mouse or by placing your cursor on the first character `!` of the goal, hitting C-space, then move to the end of the goal and type M-h g. The key combination amounts to the `set_goal` command in HOL. The result is shown below.

21

```

-----
HOL-4 [Kananaskis 9 (stdknl, built Sat Apr 26 13:42:15 2014)]

For introductory HOL help, type: help "hol";
To exit type <Control>-D (*not* quit());
-----

[loading theories and proof tools ..... ]
[closing file "/usr/local/share/HOL/tools/end-init-boss.sml"]
- ** Unicode trace now off
- > val it =
    Proof manager status: 1 proof.
    1. Incomplete goalstack:
        Initial goal:

            !x y z. x /\ y ==> z <=> x ==> y ==> z

        : proofs
- > val it = () : unit

```

Notice that what you highlighted in your sml file *does not appear in the HOL window*.

3. Our first proof step is to simplify the goal by using `STRIP_TAC` repeatedly. We type `REPEAT STRIP_TAC` under our goal, **highlight** it, and apply it to our goal by doing M-h e. This key combination is the `expand` command in HOL. We get as a result

22

```

- - - - - OK..
1 subgoal:
> val it =

    x /\ y ==> z <=> x ==> y ==> z

    : proof

```

4. We notice that the top-level operator is Boolean equality, i.e. if-and-only-if (`<=>`). The tactic `EQ_TAC` splits an if-and-only-if goal of the form `P <=> Q` into two subgoals: `P ==> Q` and `Q ==> P`. Apply `EQ_TAC` to the current subgoal by typing M-h e. Doing so produces

23

```

- OK..
2 subgoals:
> val it =

    (x ==> y ==> z) ==> x /\ y ==> z

    (x /\ y ==> z) ==> x ==> y ==> z

2 subgoals
: proof

```

5. We simplify the first subgoal,  $(x \wedge y \Rightarrow z) \Rightarrow x \Rightarrow y \Rightarrow z$ , which appears at the bottom using REPEAT STRIP\_TAC. Typing REPEAT STRIP\_TAC, highlighting it, and typing **M-h e** produces

24

```

- OK..
1 subgoal:
> val it =

    z
    -----
    0.  x /\ y ==> z
    1.  x
    2.  y
    : proof

```

6. We finish the proof of the first subgoal by using RES\_TAC, highlighting it, and typing **M-h e**.

```

- OK..
1 subgoal:
> val it =

      z
      -----
      0.  x /\ y ==> z
      1.  x
      2.  y
      : proof
- OK..

Goal proved.
[...] |- z

Goal proved.
|- (x /\ y ==> z) ==> x ==> y ==> z

Remaining subgoals:
> val it =

      (x ==> y ==> z) ==> x /\ y ==> z

      : proof

```

7. We suspect the second subgoal is likely proved in the same way. We type `REPEAT STRIP_TAC THEN RES_TAC`, highlight it, and type **M-h e**. This finishes off the proof as shown below.

```

- OK..

Goal proved.
|- (x ==> y ==> z) ==> x /\ y ==> z

Goal proved.
|- x /\ y ==> z <=> x ==> y ==> z
> val it =
  Initial goal proved.
  |- !x y z. x /\ y ==> z <=> x ==> y ==> z
  : proof

```

8. Having done the proof interactively, we now package up all the tactics into a single compound tactic. Recall that `TAC_PROOF` applied to the goal and single tactic will, if the tactic is indeed a proof, return the theorem we proved. The key is to devise a single compound tactic that does the proof correctly. We note that `EQ_TAC` produces two subgoals and that the tactics used on each subgoal were identical. This means the tactical `EQ_TAC THEN (REPEAT STRIP_TAC THEN RES_TAC)`, which applies `REPEAT STRIP_TAC THEN RES_TAC` to both subgoals generated by `EQ_TAC` should work well. We test this out by restarting the proof and supplying the compound tactics.



```

- > val it =
  Proof manager status: 2 proofs.
  2. Completed goalstack: |- !x y z. x /\ y ==> z <=> x ==> y ==> z
  1. Incomplete goalstack:
      Initial goal:

      !x y z. x /\ y ==> z <=> x ==> y ==> z

      : proofs
- > val it = () : unit

```

27

9. As before, we start with `REPEAT STRIP_TAC`, which produces just a single subgoal. We expand `REPEAT STRIP_TAC` by highlighting it and typing `M-h e` to get

```

- OK..
1 subgoal:
> val it =

  x /\ y ==> z <=> x ==> y ==> z

  : proof

```

28

10. We then finish off our proof by typing `M-h e` on `EQ_TAC THEN REPEAT STRIP_TAC THEN RES_TAC` and get

```

- OK..

Goal proved.
|- x /\ y ==> z <=> x ==> y ==> z
> val it =
  Initial goal proved.
  |- !x y z. x /\ y ==> z <=> x ==> y ==> z
  : proof

```

29

11. We finally can package up our two-step proof into a single step by combining the first `REPEAT STRIP_TAC` with `EQ_TAC THEN REPEAT STRIP_TAC THEN RES_TAC` by the tactical `THEN`. To test this out, we *back up to the initial goal* by typing `M-h B`, (**Note the capital B**). This restarts the proof.

```

- > val it =
    Initial goal:

    !x y z. x /\ y ==> z <=> x ==> y ==> z

    : proof

```

30

12. We highlight `REPEAT STRIP_TAC THEN EQ_TAC THEN REPEAT STRIP_TAC THEN RES_TAC` and type **M-h e** and see the proof done in a single step.

```

- OK..
> val it =
    Initial goal proved.
    |- !x y z. x /\ y ==> z <=> x ==> y ==> z
    : proof

```

31

13. We are now ready to use `TAC_PROOF`. The arguments are the goal with no assumptions, and the single compound tactic. We assign the theorem it returns to `exampleTheorem`.

```

val exampleTheorem =
TAC_PROOF (
  ([], ``!(x:bool) (y:bool) (z:bool). ((x /\ y) ==> z) = (x ==> (y ==> z)) ``),
  REPEAT STRIP_TAC THEN
  EQ_TAC THEN
  REPEAT STRIP_TAC THEN
  RES_TAC)

```

32

14. Highlighting the above expression and typing **M-h M-r**, (notice we are sending the highlighted region to the HOL interpreter and not asking it be evaluated as a tactic), we get

```

- > val exampleTheorem =
    |- !x y z. x /\ y ==> z <=> x ==> y ==> z
    : thm

```

33

◇

## 9.4 High-Level Proof Tactics

HOL provides high-level decision-procedures for doing proofs. An example of a powerful high-level theorem-tactic is `PROVE_TAC`. `PROVE_TAC` is a tactic that is applied to a list of theorems (similar to what you would supply to `REWRITE_TAC`). To illustrate the utility of `PROVE_TAC` consider the the previous example proved this time using `PROVE_TAC`.

**Example 9.2**

1. We set a new goal by highlighting

`!(x:bool) (y:bool) (z:bool). ((x /\ y) ==> z) = (x ==> (y ==> z))`

and typing **M-h g**. This produces a new proof goal:

```
- > val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      x y z. x y z x y z

    : proofs
- > val it = () : unit
```

34

2. We apply the tactic `PROVE_TAC[]`, notice that we provide no additional theorems, by highlighting it and typing **M-h e**.

```
- - - - - OK..
Meson search level: .....
> val it =
  Initial goal proved.
  |- x y z. x y z x y z
  : proof
```

35

◇

The above example illustrates the power of proof procedures in HOL. `PROVE_TAC` is a semi-complete proof procedure for pure first order logic. Generally speaking, when a subgoal looks straightforward to prove within the context of a list of theorems, try proving it with `PROVE_TAC` and the list of theorems. If `PROVE_TAC` is successful, it will generally be successful within a few seconds. Otherwise, it will stop on its own after a preset search depth, or you can interrupt it by typing **M-h C-c**.

## 9.5 Exercises

**Exercise 9.5.1** Do a tactic-based proof of the absorption rule, and do *not* use `PROVE_TAC`:  
[\[absorptionRule\]](#)

$$\vdash \forall p\ q. (p \Rightarrow q) \Rightarrow p \Rightarrow p \wedge q$$

**Hint:** start your proof with the tactical `REPEAT` applied to the tactic `STRIP_TAC`, i.e., `REPEAT STRIP_TAC`.

**Exercise 9.5.2** *Do a tactic-based proof of the constructive dilemma rule, and do **not** use `PROVE_TAC`: `[constructiveDilemmaRule]`*

$$\vdash \forall p\ q\ r\ s. (p \Rightarrow q) \wedge (r \Rightarrow s) \Rightarrow p \vee r \Rightarrow q \vee s$$

**Hint:** start your proof with the tactical `REPEAT` applied to the tactic `STRIP_TAC`, i.e., `REPEAT STRIP_TAC`.

**Exercise 9.5.3** *Repeat the previous exercises using `PROVE_TAC`.*

## Dealing with Assumptions in Goal-Oriented Proofs

Manipulating assumptions within the assumption list of goals is often necessary and sometimes frustrating. In this set of exercises we learn several techniques for using and manipulating terms within the assumption list of goals.

In our hand proofs, we typically refer to assumptions by their line number. This generally is a **bad idea** in HOL because there is no guarantee that the order of assumptions in proofs will be the same. Thus, we must use some other means to identify the terms we want in the assumption list. Typically, we will do this by pattern matching.

### 10.1 Rewriting and Resolution

**Rewriting with the Assumption List** The simplest cases are those where terms on the assumption list are useful for rewriting the goals. For example, suppose we have the goal  $\{\forall x : \text{num}. P(x)\} \vdash P(2)$ . We see that  $P(2)$  should be true under the assumption that  $P(x)$  is true for all values of  $x$ , i.e.,  $\forall x. P(x)$ . What we do in this case is *rewrite the goal*  $P(2)$  using the assumption  $\forall x. P(x)$ , which will prove the goal. First, we set the goal corresponding to the above.

```

- set_goal(['!(x:num).P(x)'], '(P:num->bool) (2)');
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      P 2
      -----
      !x. P x
    : proofs

```

Second, we rewrite the goal using the assumptions in the assumption list with `ASM_REWRITE_TAC`. This proves the goal and is one of the easiest situations to handle.

```

- e(ASM_REWRITE_TAC []);
OK..
> val it =
  Initial goal proved.
  [!x. P x] |- P 2 : proof

```

Finally, to clean up our proofs so they are executable in one step (setting the goal and applying the tactics, we use `TAC_PROOF` as shown below.

```

- val Example1 =
TAC_PROOF
(
  ([`!(x:num).P(x)`], `(P:num->bool) (2)`),
  (ASM_REWRITE_TAC []))
);
> val Example1 = [|x. P x] |- P 2 : thm

```

3

**Resolution** The second technique we learn deals with situations where rewriting with the assumptions is insufficient to prove the goal. In some cases, a simple application of Modus Ponens in the assumptions is enough to generate the goal within the assumptions. In these cases, a simple application of RES\_TAC is all that is needed. Suppose we have the following goal to prove, i.e.,  $\forall x y z. (x \wedge y \supset z) \Leftrightarrow (x \supset y \supset z)$ .

```

- set_goal([], `!x y z. (x /\ y) ==> z <=> (x ==> y ==> z)`);
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:
      !x y z. x /\ y ==> z <=> x ==> y ==> z
    : proofs

```

4

We use STRIP\_TAC to remove the three universal quantifiers, then we use EQ\_TAC to split the bi-conditional  $\Leftrightarrow$  into two implications.

```

- expand(REPEAT STRIP_TAC);
OK..
1 subgoal:
> val it =
  x /\ y ==> z <=> x ==> y ==> z

  : proof
- expand(EQ_TAC);
OK..
2 subgoals:
> val it =
  (x ==> y ==> z) ==> x /\ y ==> z

  (x /\ y ==> z) ==> x ==> y ==> z
  : proof

```

5

We can apply STRIP\_TAC again to simplify the goal to just  $z$  under the assumptions of  $x \wedge y \supset z$ ,  $x$ , and  $y$ . At this point, what we recognize is that all three assumptions enable us to conclude that  $z$  is true in the assumptions. If  $x$  is true then with  $x \wedge y \supset z$  we can derive  $y \supset z$ , which with the assumption  $y$  allows us to derive  $z$ . This chain of reasoning within the assumption list is invoked by RES\_TAC, as shown below. RES\_TAC is capable enough to conclude that if the goal appears in the assumptions then the proof is complete.

6

```

- expand(REPEAT STRIP_TAC);
OK..
1 subgoal:
> val it =
  z
  -----
  0. x /\ y ==> z
  1. x
  2. y
   : proof
- expand(RES_TAC);
OK..

Goal proved.
[x, y, x /\ y ==> z] |- z

Goal proved.
[] |- (x /\ y ==> z) ==> x ==> y ==> z

```

The remaining subgoal is proved in exactly the same way as the previous subgoal.

7

```

Remaining subgoals:
> val it =
  (x ==> y ==> z) ==> x /\ y ==> z
   : proof
- expand(REPEAT STRIP_TAC);
OK..
1 subgoal:
> val it =
  z
  -----
  0. x ==> y ==> z
  1. x
  2. y
   : proof
- expand(RES_TAC);
OK..

Goal proved.
[x, y, x ==> y ==> z] |- z

Goal proved.
[] |- (x ==> y ==> z) ==> x /\ y ==> z

Goal proved.
[] |- x /\ y ==> z <=> x ==> y ==> z
> val it =
  Initial goal proved.
[] |- !x y z. x /\ y ==> z <=> x ==> y ==> z : proof

```

As before, we combine both the setting of goals and the application of tactics into one operations by using **TAC\_PROOF**.

8

```

- val Example1 =
TAC_PROOF
(
  ([], ``!x y z. ((x /\ y) ==> z) <=> (x ==> y ==> z) ``),
  (REPEAT STRIP_TAC THEN
   EQ_TAC THEN
   REPEAT STRIP_TAC THEN
   RES_TAC)
);
> val Example1 = |- !x y z. x /\ y ==> z <=> x ==> y ==> z : thm

```

## 10.2 Manipulating the Assumptions with PAT\_ASSUM

Consider the following goal:  $\{\neg p \vee q, p\} \vdash q$ . We set the goal with `set_goal` as shown below.

```

- set_goal(['~p \/ q', 'p:bool', 'q:bool']);
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      q
      -----
    0. p
    1. ~p \/ q

: proofs

```

9

We recognize that  $\neg p \vee q$  is equivalent to  $p \supset q$ . We note that  $p$  is also part of the assumptions, so we would like to derive  $q$  and be done with our proof. If we apply `RES_TAC` to the assumption list, here is what happens.

```

- e(RES_TAC);
OK..
1 subgoal:
> val it =

  q
  -----
  0. p
  1. ~p \/ q

: proof

```

10

We see from above that *nothing* happened! Basically, `RES_TAC` is not smart enough to recognize that  $\neg p \vee q$  is equivalent to  $p \supset q$ . What we need to do is *replace*  $\neg p \vee q$  with  $p \supset q$  in the assumption list and then use `RES_TAC` to complete the proof.

The primary question is how do we “grab on” to terms in the assumption list? We could refer to  $\neg p \vee q$  by its index in the assumption list, i.e., assumption 1. This technique is very brittle. If for some reason the next time the order of assumptions is somehow different, our proof will no longer work.

Instead of referring to assumptions by their place on the assumption list, we refer to them by their form or *pattern*. The `PAT_ASSUM` tactical is an effective way for manipulating terms on the assumption list. `PAT_ASSUM` take two arguments:

1. a HOL term that is the pattern of the assumption to be manipulated, and
2. a theorem-tactic, i.e., a function that takes a theorem (a theorem corresponding to a term on the assumption list specified by the pattern) and returns a tactic as a result.

For our example above, the pattern we use is `~A \/ B`. Note, we could have just as easily used the term we want to manipulate itself as the pattern, i.e., `~p \/ q`. However, to illustrate the pattern matching capabilities of `PAT_ASSUM`, we choose `~A \/ B`.

Next, we have to come up with a function that is a theorem-tactic. In this case, assume that we have a theorem `th` that corresponds to `[.] |- ~A \/ B`. What we want to do is (1) convert `th` into a theorem of the form `A ==> B`, and (2) add the theorem to the assumption list. We do (1) by `DISJ_IMP` followed



by `REWRITE_RULE []` to simplify the double negated  $A$ , and then we accomplish (2) by `ASSUME_TAC` applied to the rewritten theorem.

The session below shows how we develop the theorem to be added to the assumption list.

```
- val th = ASSUME ``~A \ / B``;
> val th = [~A \ / B] |- ~A \ / B : thm
- DISJ_IMP th;
> val it = [~A \ / B] |- ~~A ==> B : thm
- REWRITE_RULE [] (DISJ_IMP th);
> val it = [~A \ / B] |- A ==> B : thm
```

11

We finally build anonymous function that is our theorem tactic. The theorem we want `ASSUME_TAC` to add is `REWRITE_RULE [] (DISJ_IMP th)`, where `th` is the parameter representing the theorem to which the theorem-tactic is applied. Recall that lambda functions of the form  $\lambda x.t$  are represented in ML as `fn x => t`. Thus, the function that is our theorem tactic is

```
fn th => ASSUME_TAC (REWRITE_RULE [] (DISJ_IMP th))
```

The expression we use is

```
PAT_ASSUM ``~A \ / B``
  (fn th => ASSUME_TAC (REWRITE_RULE [] (DISJ_IMP th))).
```

The result is shown in the session below. First, we undo by backup up from the unproductive `RES_TAC` step. Then we apply `PAT_ASSUM`. We see that the result is  $\sim p \ \backslash / \ q$  is replace by  $p \ ==> \ q$ .

```
- b();
> val it =
  Initial goal:

  q
  -----
  0. p
  1. ~p \ / q
  : proof
- e(PAT_ASSUM ``~A \ / B`` (fn th => ASSUME_TAC (REWRITE_RULE [] (DISJ_IMP th))));
OK..
1 subgoal:
> val it =

  q
  -----
  0. p
  1. p ==> q
  : proof
```

12

At this point, we have a simple case where `RES_TAC` can handle the rest.

```
- e(RES_TAC);
OK..

Goal proved.
[p, p ==> q] |- q
> val it =
  Initial goal proved.
  [p, ~p \ / q] |- q : proof
```

13

As always, we package everything up inside of TAC\_PROOF.

```
- val Example3 =
TAC_PROOF
(
  ([`~p \ / q`, `p:bool`, `q:bool`),
  (PAT_ASSUM ``A \ / B`` (fn th => ASSUME_TAC (REWRITE_RULE [] (DISJ_IMP th))) THEN
  RES_TAC)
);
> val Example3 = [p, ~p \ / q] |- q : thm
```

14

## 10.3 More Examples

### Dealing with Contrapositives in the Assumption List

```
- set_goal([`p ==> q`, `~q`, `~p`]);
> val it =
Proof manager status: 2 proofs.
2. Completed goalstack: [p, ~p \ / q] |- q
1. Incomplete goalstack:
  Initial goal:

  ~p
  -----
  0. ~q
  1. p ==> q

: proofs
```

15

The plan is to do the following:

1. Use ``A ==> B`` as the pattern for theorem *th*
2. Build a theorem-tactical that does the following for *th* of the form  $A \supset B$ :
  - a. Convert  $A \supset B$  to  $\neg A \vee B$  by applying `IMP_ELIM`.
  - b. Rewrite *once*  $\neg A \vee B$  using the theorem `DISJ_SYM` ( $\vdash \forall A B. A \vee B \Leftrightarrow B \vee A$ ) to get  $B \vee \neg A$ .
  - c. Convert  $B \vee \neg A$  to its equivalent form  $\neg B \supset \neg A$  using the `DISJ_IMP` inference rule.
  - d. Add the resulting theorem  $\neg B \supset \neg A$  to the assumption list using `ASSUME_TAC`. The theorem-tactic is

$$\lambda th. ASSUME\_TAC(DISJ\_IMP(ONCE\_REWRITE\_RULE[DISJ\_SYM](IMP\_ELIM th)))$$

In HOL, we write the above as:

```
fn th =>
  ASSUME_TAC
  (DISJ_IMP (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))
```

Everything is put together as shown below.

```

- e(PAT_ASSUM
  ``A ==> B``
  (fn th =>
    ASSUME_TAC
    (DISJ_IMP
      (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))));
OK..
1 subgoal:
> val it =

  ~p
  -----
  0.  ~q
  1.  ~q ==> ~p
  : proof

```

16

We see from the assumption list that if we apply RES\_TAC the proof will be finished.

```

- e(RES_TAC);
OK..

Goal proved.
[~q, ~q ==> ~p] |- ~p
> val it =
  Initial goal proved.
  [~q, p ==> q] |- ~p : proof

```

17

As always, we package up everything into one application of TAC\_PROOF.

```

- val Example4 =
  TAC_PROOF
  (
    ([``p ==> q``, ``~q``, ``~p``], ``~p``),
    (PAT_ASSUM
      ``A ==> B``
      (fn th =>
        ASSUME_TAC
        (DISJ_IMP
          (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))) THEN
      RES_TAC)
  );
> val Example4 = [~q, p ==> q] |- ~p : thm

```

18

**Reasoning with Contrapositives and Universally Quantified Statements** In this example we have the following goal:  $\{\forall x : num. P(x) \supset Q(x), \neg Q(5)\} \vdash \neg P(5)$ .

```

- set_goal(['!(x:num).(P(x) ==> Q(x))', '~(Q:num->bool) (5)'],
          '~(P:num->bool) (5)');
> val it =
  Proof manager status: 3 proofs.
  3. Completed goalstack: [p, ~p \ / q] |- q
  2. Completed goalstack: [~q, p ==> q] |- ~p
  1. Incomplete goalstack:
      Initial goal:

      ~P 5
      -----
      0.  ~Q 5
      1.  !x. P x ==> Q x

      : proofs

```

19

What we need to do is specialize  $x$  in  $\forall x.P(x) \supset Q(x)$  to 5, take the contrapositive, and do resolution to prove the goal. First, we specialize  $x$  to 5 using `PAT_ASSUM` and replace  $\forall x.P(x) \supset Q(x)$  with  $P(5) \supset Q(5)$ .

```

- e(PAT_ASSUM '!x.t' (fn th => (ASSUME_TAC (SPEC '5' th))));
<<HOL message: inventing new type variable names: 'a>>
OK..
1 subgoal:
> val it =

  ~P 5
  -----
  0.  ~Q 5
  1.  P 5 ==> Q 5

  : proof

```

20

As we have done before, we take the contrapositive of  $P(5) \supset Q(5)$  using `PAT_ASSUM`.

```

- e(PAT_ASSUM
    'A ==> B'
    (fn th =>
      ASSUME_TAC
      (DISJ_IMP
       (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))));
OK..
1 subgoal:
> val it =

  ~P 5
  -----
  0.  ~Q 5
  1.  ~Q 5 ==> ~P 5

  : proof

```

21

Finally, we finish the proof using `RES_TAC`.

```

- e (RES_TAC);
OK..

Goal proved.
[~Q 5, ~Q 5 ==> ~P 5] |- ~P 5

Goal proved.
[~Q 5, P 5 ==> Q 5] |- ~P 5
> val it =
  Initial goal proved.
  [!x. P x ==> Q x, ~Q 5] |- ~P 5 : proof

```

22

As always, we package up the entire proof within TAC\_PROOF.

```

- val Example5 =
  TAC_PROOF
  (
    ([!!(x:num). (P(x) ==> Q(x)) '', ~~(Q:num->bool) (5) ''), ~~(P:num->bool) (5) ''),
    (PAT_ASSUM !!x.t'' (fn th => (ASSUME_TAC (SPEC ''5'' th))) THEN
     PAT_ASSUM
     ''A ==> B''
     (fn th =>
      ASSUME_TAC
      (DISJ_IMP
       (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))) THEN
      RES_TAC)
    );
<<HOL message: inventing new type variable names: 'a>>
> val Example5 = [!x. P x ==> Q x, ~Q 5] |- ~P 5 : thm

```

23

**Tightening Patterns When Needed** Sometimes, more than one term in the assumption list matches the supplied pattern. In these cases we can be more specific with the term we supply to PAT\_ASSUM to match terms against.

For example, consider the following case where we have two terms with the form  $A \implies B$ :

```

- set_goal([!!(x:'a. (P:'a -> bool) x) ==> r'', ~p ==> q'', ~~r'',
           ~p:bool''], ~?x:'a. ~P x'');
> val it =
  Proof manager status: 5 proofs.
  5. Completed goalstack: [p, ~p \ / q] |- q
  4. Completed goalstack: [~q, p ==> q] |- ~p
  3. Completed goalstack: [!x. P x ==> Q x, ~Q 5] |- ~P 5
  2. Completed goalstack: [!x. P x ==> Q x, ~Q 5] |- ~P 5
  1. Incomplete goalstack:
    Initial goal:

    ?x. ~P x
    -----
    0. p
    1. ~r
    2. p ==> q
    3. (!x. P x) ==> r

    : proofs

```

24

The following does what we want in that `''A ==> B''` gets us the right implication we want to manipulate, i.e., replace  $(!x. P x) \implies r$  with  $\sim r \implies \sim !x. P x$ .

```

e (PAT_ASSUM
  ``A ==> B``
  (fn th =>
    ASSUME_TAC
    (DISJ_IMP
      (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))));
OK..
1 subgoal:
> val it =

  ?x. ~P x
  -----
  0.  p
  1.  ~r
  2.  p ==> q
  3.  ~r ==> ~!x. P x
  : proof

```

However, suppose the assumption list has a slightly different order, as shown below. If we do the same PAT\_ASSUM operation, we get:

```

- set_goal(['`p ==> q`,`(!x:'a.(P:'a -> bool) x) ==> r`,` ``~r`,`
          ``p:bool`,` ``?x:'a.~P x`]);
> val it =
  Proof manager status: 6 proofs.
  6. Completed goalstack: [p, ~p \ / q] |- q
  5. Completed goalstack: [~q, p ==> q] |- ~p
  4. Completed goalstack: [!x. P x ==> Q x, ~Q 5] |- ~P 5
  3. Completed goalstack: [!x. P x ==> Q x, ~Q 5] |- ~P 5
  2. Incomplete goalstack:
    Initial goal:

    ?x. ~P x
    -----
    0. p
    1. ~r
    2. p ==> q
    3. (!x. P x) ==> r

    Current goal:

    ?x. ~P x
    -----
    0. p
    1. ~r
    2. p ==> q
    3. ~r ==> ~!x. P x

  1. Incomplete goalstack:
    Initial goal:

    ?x. ~P x
    -----
    0. p
    1. ~r
    2. (!x. P x) ==> r
    3. p ==> q

    : proofs
- e(PAT_ASSUM
  ``A ==> B``
  (fn th =>
    ASSUME_TAC
    (DISJ_IMP
      (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))));
OK..
1 subgoal:
> val it =

  ?x. ~P x
  -----
  0. p
  1. ~r
  2. (!x. P x) ==> r
  3. ~q ==> ~p
  : proof

```

We see from the above that the wrong implication was manipulated by PAT\_ASSUM. To fix this, we tighten up our pattern to the exact term we wish to select,  $(!x. P\ x) ==> r$ . After backing up from selecting the wrong term and tightening up our term selection, we get:

27

```

- b();
e (PAT_ASSUME
  ``(!x:'a.P x) ==> r``
  (fn th =>
    ASSUME_TAC
    (DISJ_IMP
      (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))));
> val it =
  Initial goal:

  ?x. ~P x
  -----
  0. p
  1. ~r
  2. (!x. P x) ==> r
  3. p ==> q
  : proof
- OK..
1 subgoal:
> val it =

  ?x. ~P x
  -----
  0. p
  1. ~r
  2. p ==> q
  3. ~r ==> ~!x. P x
  : proof

```

We apply RES\_TAC to deduce  $\neg\forall x.P(x)$  in the assumptions.

28

```

- e (RES_TAC);
OK..
1 subgoal:
> val it =

  ?x. ~P x
  -----
  0. p
  1. ~r
  2. p ==> q
  3. ~r ==> ~!x. P x
  4. ~!x. P x
  5. q
  : proof

```

At this point, you may remember the relation  $\exists x.\neg P(x) = \neg\forall x.P(x)$ . We can find this theorem by consulting bool theory in the browser-compatible interface to the HOL Reference Manual. Consult Appendix 20 to find the location of this interface either locally or on-line. The theorem we want is NOT\_FORALL\_THM, as shown below.

29

```

- NOT_FORALL_THM;
> val it = [] |- !P. ~(!x. P x) <=> ?x. ~P x : thm

```

We can do the proof in several ways. One way is to rewrite the goal using NOT\_FORALL\_THM. The second way is to rewrite the assumption.



**Rewriting the Goal** To rewrite the goal we need to reverse the order of terms in NOT\_FORALL\_THM. This is done by specializing all the variables in NOT\_FORALL\_THM and using the SYM inference rule the result.

```
- SYM (SPEC_ALL NOT_FORALL_THM);
> val it = [] |- (?x. ~P x) <=> ~!x. P x : thm
```

30

With the above theorem, we rewrite the goal and finish the proof using ASM\_REWRITE\_TAC.

```
- e(ASM_REWRITE_TAC [SYM (SPEC_ALL NOT_FORALL_THM)]);
OK..

Goal proved.
[~!x. P x] |- ?x. ~P x

Goal proved.
[p, ~r, p ==> q, ~r ==> ~!x. P x] |- ?x. ~P x
> val it =
  Initial goal proved.
  [p, ~r, p ==> q, (!x. P x) ==> r] |- ?x. ~P x : proof
```

31

As always, we bundle up everything within TAC.PROOF.

```
- val Example6 =
TAC_PROOF
(
  ([~p ==> q'', ~(!x:'a.(P:'a -> bool) x) ==> r'', ~r'', ~p:bool''],
   ~?x:'a.~P x''),
  (PAT_ASSUM
   ~(!x:'a.P x) ==> r''
   (fn th =>
    ASSUME_TAC
    (DISJ_IMP
     (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))) THEN
    RES_TAC THEN
    ASM_REWRITE_TAC [SYM (SPEC_ALL NOT_FORALL_THM)]))
);
> val Example6 = [p, ~r, p ==> q, (!x. P x) ==> r] |- ?x. ~P x : thm
```

32

**Rewriting the Assumptions** The second approach is to rewrite the assumptions using NOT\_FORALL\_THM. This approach requires a bit of precision when dealing with the assumptions—but it is good practice for us. Our approach is as follows:

1. We use PAT\_ASSUM applied to `~!x. P x` to specify that it is the theorem `th` that we will use in our theorem-tactic.
2. We will rewrite `~!x. P x` to `?x. ~P x` by REWRITE\_RULE [NOT\_FORALL\_ASSUM] `th`.
3. The above theorem is put into the assumption list by ASSUME\_TAC.
4. The function that is the theorem-tactic is  $\lambda th. \text{ASSUME\_TAC } (\text{REWRITE\_RULE } [\text{NOT\_FORALL\_THM}] \text{ th})$ . This is written in ML as

```
fn th => ASSUME_TAC (REWRITE_RULE [NOT_FORALL_THM] th)
```

The result of doing the above is shown below.

```

- restart();
> val it =
  Initial goal:

  ?x. ~P x
  -----
  0. p
  1. ~r
  2. (!x. P x) ==> r
  3. p ==> q
  : proof
- e(PAT_ASSUM
  ``~!(x:'a).P x``
  (fn th =>
    ASSUME_TAC
    (REWRITE_RULE [NOT_FORALL_THM] th))
  );
OK..
1 subgoal:
> val it =

  ?x. ~P x
  -----
  0. p
  1. ~r
  2. p ==> q
  3. ~r ==> ~!x. P x
  4. q
  5. ?x. ~P x
  : proof

```

33

We see the goal in the assumption list, which means we can finish the proof by `ASM_REWRITE_TAC`.

```

- e(ASM_REWRITE_TAC []);
OK..

Goal proved.
[?x. ~P x] |- ?x. ~P x

Goal proved.
[~!x. P x] |- ?x. ~P x

Goal proved.
[p, ~r, p ==> q, ~r ==> ~!x. P x] |- ?x. ~P x
> val it =
  Initial goal proved.
  [p, ~r, p ==> q, (!x. P x) ==> r] |- ?x. ~P x : proof

```

34

Finally, we wrap everything within `TAC_PROOF` to prove the desired theorem in one step.

35

```

- val Example6 =
TAC_PROOF
(
  ([`p ==> q`, `(!x:'a.(P:'a -> bool) x) ==> r`, `~r`, `p:bool`],
  ``?x:'a.~P x``),
(PAT_ASSUM
  ``(!x:'a.P x) ==> r``
  (fn th =>
    ASSUME_TAC
    (DISJ_IMP
      (ONCE_REWRITE_RULE [DISJ_SYM] (IMP_ELIM th)))) THEN
  RES_TAC THEN
  PAT_ASSUM
  ``~!(x:'a).P x``
  (fn th =>
    ASSUME_TAC
    (REWRITE_RULE [NOT_FORALL_THM] th) THEN
    ASM_REWRITE_TAC []))
);
> val Example6 = [p, ~r, p ==> q, (!x. P x) ==> r] |- ?x. ~P x : thm

```

**Combining Forward Proofs with the Assumption List** Sometimes the simplest thing is to combine forward proofs with backwards proofs. Consider the goal  $\{p \supset q, q \supset r\} \text{ ?- } p \supset r$ . There are two ways of dealing with this. The first way strips the implication in the goal and moves  $q$  into the assumption list followed by repeated resolutions to generate the goal in the assumption list. This is illustrated below.

```

- set_goal(['`p ==> q`', '`q ==> r`'], '`p ==> r`');
> val it =
  1. Incomplete goalstack:
      Initial goal:

          p ==> r
          -----
          0. q ==> r
          1. p ==> q

      : proofs
- e(STRIP_TAC);
OK..
1 subgoal:
> val it =

    r
    -----
    0. q ==> r
    1. p ==> q
    2. p
    : proof
- e(RES_TAC);
OK..
1 subgoal:
> val it =

    r
    -----
    0. q ==> r
    1. p ==> q
    2. p
    3. q
    : proof
- e(RES_TAC);
OK..

Goal proved.
[q, q ==> r] |- r

Goal proved.
[p, p ==> q, q ==> r] |- r
> val it =
  Initial goal proved.
  [p ==> q, q ==> r] |- p ==> r : proof

- val Example7 =
TAC_PROOF
([['`p ==> q`', '`q ==> r`'], '`p ==> r`'],
 (STRIP_TAC THEN
  REPEAT RES_TAC)
);
> val Example7 = [p ==> q, q ==> r] |- p ==> r : thm

```

Another approach to the above is to add the results of a forward proof to the assumption list. Specifically, the theorem derived by

$$\text{IMP\_TRANS (ASSUME ``p ==> q``) (ASSUME ``q ==> r``)}$$

is  $\{p \supset q, q \supset r\} \vdash p \supset r$ . As the sequent's assumptions are contained in the assumptions on the assumption list, we can add  $p \supset r$  to the assumptions. Afterward, we use `ASM_REWRITE_TAC` to finish the proof. The session below shows this approach to the proof. Notice that `RES_TAC` does not know how to use transitivity on the implications in the assumptions.

37

```

- set_goal(['`p ==> q`,`q ==>r`'], '`p ==> r`');
> val it =
  1. Incomplete goalstack:
      Initial goal:
        p ==> r
      -----
        0. q ==> r
        1. p ==> q

      : proofs
- e(RES_TAC);
OK..
1 subgoal:
> val it =

  p ==> r
  -----
    0. q ==> r
    1. p ==> q
      : proof
- b();
> val it =
  Initial goal:

  p ==> r
  -----
    0. q ==> r
    1. p ==> q
      : proof
- IMP_TRANS (ASSUME ``p ==> q``) (ASSUME ``q ==> r``);
> val it = [p ==> q, q ==> r] |- p ==> r : thm
- e(ASSUME_TAC(IMP_TRANS (ASSUME ``p ==> q``) (ASSUME ``q ==> r``)));
OK..
1 subgoal:
> val it =

  p ==> r
  -----
    0. q ==> r
    1. p ==> q
    2. p ==> r
      : proof
- e(ASM_REWRITE_TAC []);
OK..

Goal proved.
[p ==> r] |- p ==> r
> val it =
  Initial goal proved.
  [p ==> q, q ==> r] |- p ==> r : proof
- val Example7 =
TAC_PROOF
(
  ([`p ==> q`,`q ==>r`], '`p ==> r`'),
  (ASSUME_TAC(IMP_TRANS (ASSUME ``p ==> q``) (ASSUME ``q ==> r``)) THEN
   ASM_REWRITE_TAC [])
);
> val Example7 = [p ==> q, q ==> r] |- p ==> r : thm

```

## 10.4 Exercises

**Exercise 10.4.1** Prove the following goal *without* using *PROVE\_TAC*. Your final solution must be executable in a single step using *TAC\_PROOF*.

```
set_goal
([`!x:'a. P(x) ==> M(x)`, `(P:'a->bool) (s:'a)`,
 `(M:'a->bool) (s:'a)`];
```

Save the proved theorem as

[\[problem1\\_thm\]](#)

$\vdash M\ s$

**Exercise 10.4.2** Prove the following goal *without* using *PROVE\_TAC*. Your final solution must be executable in a single step using *TAC\_PROOF*.

```
set_goal([`p /\ q ==> r`, `r ==> s`, `~s`, `p ==> ~q`];
```

Save the proved theorem as

[\[problem2\\_thm\]](#)

$\vdash p \Rightarrow \neg q$

**Exercise 10.4.3** Prove the following goal *without* using *PROVE\_TAC*. Your final solution must be executable in a single step using *TAC\_PROOF*.

```
set_goal([`~(p /\ q)`, `~p ==> r`, `~q ==> s`, `r /\ s`];
```

(Hint: you might want to look up DeMorgan's theorem in HOL.)

Save the proved theorem as

[\[problem3\\_thm\]](#)

$\vdash r \vee s$

# Defining Theories, Types, and Functions in HOL

---

At this point, we have learned the basic proof techniques in HOL. Specifically, we have learned to use forward inference rules and tactics for goal-oriented proofs. In this chapter, we show how to use existing theories in HOL, and how new theories, types, and definitions are added to HOL. The HOL System Description [HOL, 2015a], which is one of the manuals included in the HOL distribution, has a fuller explanation of types in HOL. For quick reference, consult the browser interface to the HOL Reference manual [HOL, 2015c], which is found in the *help* folder of the the HOL distribution.

## 11.1 Defining New Theories in HOL

In this section, we learn how to (1) extend HOL by defining new theories, and (2) naming and structuring our theory files to enable easy reproduction and updating of theories due to changes and updates. Chapter 6.3 in the HOL System Description manual [HOL, 2015a] gives a detailed description Script and Theory files in HOL, as well as the use of `Holmake` to regenerate theories.

When creating new theories in HOL, exercising a little planning up front avoids a lot of frustration in the end. Theories in HOL are developed in two modes:

1. interactive theory construction, where you work back and forth between your source code file and the HOL interpreter, and
2. theory update and rebuild, where you rebuild your theories as a result of modifying or adding existing or new theories. Theory rebuilds are done using the HOL function **Holmake**.

Sometimes, after an arduous and long series of interactive sessions creating a theory, particularly where there are dependencies among theories, incompatibilities arise when rebuilding the modified collection of HOL theories. In many ways, this is similar to developing a proof interactively step-by-step using tactics and then having the composite compound tactic fail, except at the level of whole theories.

To avoid such frustrations, we adopt a theory development style that incorporates:

1. theory source code conventions that anticipate the needs of theory updates and rebuilds, and
2. frequent theory rebuilds during theory development to unearth rebuilding problems early.

### Source Code Conventions for Holmake

Using `Holmake` requires following several source code conventions.

1. Source code file names must include the theory name followed by “Script” If we wish to define a new theory, say *myTheory*, our source code file must be named *myScript.sml*. Your file name must **not** include “Theory”, as HOL creates files with “Theory” in their names when rebuilding theories.

2. HOL theories are ML structures. This means all *Script* files must begin with

```
structure myScript = struct
```

and end with

```
end
```

3. When using *Holmake*, the following theories, libraries, and structures must be opened.

```
open HolKernel boolLib Parse bossLib
```

4. If the theory depends on other theories, say *ancestor1Theory* and *ancestor2Theory*, they must be opened.

```
open ancestor1Theory ancestor2Theory
```

5. A declaration saying what follows is intended to be a newly defined theory called *myTheory*.

```
val _ = new_theory "my"
```

6. After all the code for definitions and proofs, the theory must be exported. When rebuilding theories with *Holmake*, it is convenient to print the contents of theories after they are built, to track the progress of the rebuilding. We do this with the following code.

```
val _ = export_theory ();  
val _ = print_theory "-";
```

In the following example, we illustrate the application of the above conventions when defining an example theory *exTypeTheory*.

### Example 11.1

Do the following:

1. Create a file `exTypeScript.sml`. Put the following code into the file.

```
structure exTypeScript = struct  
  
  open HolKernel boolLib Parse bossLib  
  open listTheory TypeBase  
  
  val _ = new_theory "exType";  
  
  val _ = export_theory ();  
  val _ = print_theory "-";  
  
end
```

2. In a terminal, go to the subdirectory containing `exTypeScript.sml` and execute `Holmake cleanAll` followed by `Holmake`. You should see what appears in the HOL session box below.



```

$ Holmake cleanAll
$ Holmake
Holmake: ./usr/local/share/HOL/bin/Holmake: Analysing exTypeScript.sml
/usr/local/share/HOL/bin/Holmake: Trying to create directory .HOLMK for dependency files
Compiling exTypeScript.sml
Linking exTypeScript.uo to produce theory-builder executable
<<HOL message: Created theory "exType">>
Exporting theory "exType" ... done.
Theory "exType" took 0.000s to build
Theory: exType

Parents:
  list

/usr/local/share/HOL/bin/Holmake: Analysing exTypeTheory.sml
/usr/local/share/HOL/bin/Holmake: Analysing exTypeTheory.sig
Compiling exTypeTheory.sig
Compiling exTypeTheory.sml
$

```

## 11.2 Using Existing Theories and Types in HOL

Once we have introduced the structure of a new theory into HOL and compiled it with `Holmake`, we can work interactively to develop the new theory. As an example, we illustrate how we can develop *exTypeTheory* interactively, and then recompile it using `Holmake` after each change. As part of our example development, we will introduce additional proof techniques, such as *structural induction* and *case analysis*.

### Example 11.2

At this point, we have defined the structure of *exTypeTheory* and successfully built it using `Holmake`. This means that HOL “knows” about *exTypeTheory* and we can load and open the it and extend the theory interactively.

Do the following:

1. Using Emacs, open *exTypeScript.sml* and start a HOL session.
2. In your source code, right after the `structure` declaration, add the following code.

```

structure exTypeScript = struct

(* ==== Interactive mode: within a comment so it isn't executed by Holmake ====
map load ["listTheory", "TypeBase", "exTypeTheory"];
open listTheory TypeBase exTypeTheory
===== end Interactive mode =====*)

```

What the above addition does is embed the commands to interactively load and open *listTheory*, *TypeBase*, and *exTypeTheory* within a multi-line comment. This enables *exTypeScript.sml* to be compiled by `Holmake` without causing an error due to using the `load` command or attempting to open *exTypeTheory* before it is defined.

3. Highlight the two lines containing `map ["listTheory", "TypeBase", "exTypeTheory"];` and `open listTheory TypeBase exTypeTheory`. Either copy them into the HOL interpreter or execute them using [M-h M-r](#).

4. What you should see is in the session box below.

2

```

- map load ["listTheory", "TypeBase", "exTypeTheory"];
> val it = [(), (), ()] : unit list
- open listTheory TypeBase exTypeTheory;
> type typeBase = typeBase
  type term = term
  type hol_type = hol_type
  type shared_thm = shared_thm
  type tyinfo = tyinfo
  type thm = thm

.... many pages of output omitted ....

- <<HOL message: inventing new type variable names: 'a>>

```

What the above session illustrates is that to use previously defined theories you must do two steps:

1. Load the theory using `load`, where the name of the theory is within string quotes, and
2. Open the theory using `open`, where the name of the theory appears **without** string quotes.

In our example, notice that we using the higher-order ML function `map` applied to `load` and the list of theory-string names, `["listTheory", "TypeBase", "exTypeTheory"]`. Also, note that `open` is just applied to the sequence of theory names separated by spaces, i.e., `open listTheory TypeBase exTypeTheory`.

To check if we have broken the script file in any way, we can go to a terminal and execute `Holmake` in the same subdirectory as our file and make sure it compiles correctly. ◇

At this point, after loading the theory we are extending and the theories, libraries, and structure, which our theory uses, we continue adding definitions of types or functions, and proving their properties. All of these additions are added to the script file **after** `val _ = new_theory "<theory name>"` and **before** `val _ = export_theory()`.

## 11.3 Adding New Definitions to HOL

In our next example, we show how to extend HOL by defining a new function on lists, and prove a theorem about its properties. For a thorough description, read Chapter 4.5 of *The HOL System DESCRIPTION*, [HOL, 2015a].

### Example 11.3

At this point, we have loaded and opened `listTheory`, `TypeBase`, and `exTypeTheory`. We now define our own version of the list append function. Our function is named `APP`. We do this using the function `Define` as described in Chapter 4.5 of *The HOL System DESCRIPTION*, [HOL, 2015a]. As normal, we define `APP` using pattern matching. Each clause corresponds to the definition of `APP` when the first list is empty or when the first list is non-empty. Note, in HOL the list constructor `CONS` is `::`, i.e., a double colon.

If you have not done so already, open `exTypeScript.sml` in Emacs, start a HOL session, load and open `listTheory`, `TypeBase`, and `exTypeTheory`. Do the following.

1. Add the following code (notice that all our additions are after `val _ = new_theory "exType"`).

```
val _ = new_theory "exType";

(* ==== start here ====
val APP_def =
Define
'(APP [] (l:'a list) = l) /\
 (APP (h::(l1:'a list)) (l2:'a list) = h::(APP l1 l2)) '
==== end here ==== *)
```

2. Highlight the code in-between the multi-line comment `(* ==== start here ====` and `==== end here ==== *)`, and either copy it into the HOL interpreter or send the region to the interpreter using [M-h M-r](#). What you should see is shown in the boxed HOL session below.

```
- val APP_def =
Define
'(APP [] (l:'a list) = l) /\
 (APP (h::(l1:'a list)) (l2:'a list) = h::(APP l1 l2))';
Definition has been stored under "APP_def"
> val APP_def =
|- (!l. APP [] l = l)  !h l1 l2. APP (h::l1) l2 = h::APP l1 l2
: thm
```

3

There are a few things to notice about the above definition.

1. Definitions in HOL are stored by default under the name `<name>_def`. Thus, while developing your theory interactively, you should follow the same convention, so when you want to use the definition of the function later on, you are using the stored name consistently.
2. The definition of `APP` relies on pattern matching over lists. The type declaration `l:'a list` means `l` is a polymorphic list of elements of type `'a`, where `'a` is a type variable.
3. We initially put our definition within a multi-line comment to allow recompiling `exTypeTheory` using `Holmake`, prior to fully debugging our extension.

In this case, our definition of `APP` did not cause any problems, so we place it outside the scope of the multi-line comment, and verify that everything compiles correctly using `Holmake`.

Do the following.

1. Place your definition of `APP` outside the scope of the multi-line comment. Your code should look like the code below.

```
val APP_def =
Define
'(APP [] (l:'a list) = l) /\
 (APP (h::(l1:'a list)) (l2:'a list) = h::(APP l1 l2)) '

(* ==== start here ====
==== end here ==== *)
```

2. Execute `Holmake` in a terminal in the same subdirectory containing `exTypeScript.sml`. The output should match what appears below.

```

$ Holmake
/usr/local/share/HOL/bin/Holmake: Analysing exTypeScript.sml
Compiling exTypeScript.sml
Linking exTypeScript.uo to produce theory-builder executable
<<HOL message: Created theory "exType">>
Saved definition __ "APP_def"
Exporting theory "exType" ... done.
Theory "exType" took 0.428s to build
Theory: exType

Parents:
  list

Term constants:
  APP  : list -> list -> list

Definitions:
  APP_def
    |- (!l. APP [] l = l) !h l1 l2. APP (h::l1) l2 = h::APP l1 l2

/usr/local/share/HOL/bin/Holmake: Analysing exTypeTheory.sig
Compiling exTypeTheory.sig
/usr/local/share/HOL/bin/Holmake: Analysing exTypeTheory.sml
Compiling exTypeTheory.sml
$

```

Notice that now the print out of *exTypeTheory* includes *APP* as a term constant along with its type signature, and its definition appears under Definitions.  $\diamond$

## 11.4 Proofs Using Structural Induction

One of the most useful features of HOL are the structural induction tactics. In combination with *ASM\_REWRITE\_TAC* or *PROVE\_TAC*, the induction tactics significantly simplify proving theorems. Our next example is an illustration.

### Example 11.4

Suppose we wish to prove that *APP* is associative, i.e.,

$$\vdash \forall l_1 \ l_2 \ l_3. \text{APP} (\text{APP} \ l_1 \ l_2) \ l_3 = \text{APP} \ l_1 (\text{APP} \ l_2 \ l_3),$$

where  $l_1$ ,  $l_2$ , and  $l_3$  are polymorphic lists.

Do the following.

1. Enter the following code after the definition of *APP*.

```

(*****
(* Proof that APP is associative. *)
*****
(* ===== start here =====
set_goal([],
  '!(l1:'a list)(l2:'a list)(l3:'a list).
  (APP(APP l1 l2) l3) = (APP l1 (APP l2 l3))'
===== end here ===== *)

```

Notice that we are doing our proof development within a multi-line comment so that we can always execute *Holmake* without problems.

2. Executing `set_goal` in HOL gives the following result.

```

- > val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !l1 l2 l3. APP (APP l1 l2) l3 = APP l1 (APP l2 l3)

: proofs

```

5

Given that `APP` is defined recursively on its first list argument, we use structural induction on `l1` in the goal. We do this using the function `Induct_on`l1``. Notice that ``l1`` is a HOL term fragment, and `l1` is not within double back-quotes.

Do the following.

1. Extend your interactive proof with `Induct_on`l1``, as shown below.

```

(* ==== start here ====
set_goal([],
  '!(l1:'a list)(l2:'a list)(l3:'a list).
  (APP(APP l1 l2) l3) = (APP l1 (APP l2 l3))'
  Induct_on `l1`
  ==== end here ==== *)

```

2. Apply the tactic `Induct_on`l1`` by highlighting it and sending it to HOL by [M-h e](#), or by copying and pasting it into HOL and applying the expand function `e` to it. You should see the following result.

```

- OK..
2 subgoals:
> val it =

  !h l2 l3. APP (APP (h::l1) l2) l3 = APP (h::l1) (APP l2 l3)
  -----
  !l2 l3. APP (APP l1 l2) l3 = APP l1 (APP l2 l3)

  !l2 l3. APP (APP [] l2) l3 = APP [] (APP l2 l3)

2 subgoals
: proof

```

6

The first subgoal is the base case, where the empty list is substituted for `l1`. The second subgoal is the inductive step, where `h::l1` is substituted for `l1` and the inductive hypothesis `!l2 l3. APP (APP l1 l2) l3 = APP l1 (APP l2 l3)` is in the assumption list. Looking at the subgoals, we see that the forms of both the base case and inductive-step match the form of the definition of `APP` in `APP_def`. We use both `APP_def` and the inductive hypothesis in the assumptions of the inductive step, to rewrite the subgoals.

Do the following.

1. Extend your interactive proof with `ASM_REWRITE_TAC [APP_def]`, as shown below. Note the use of the tactical `THEN` after `Induct_on`l1``.

```
(* ==== start here ====
set_goal([],
  '!(l1:'a list)(l2 : 'a list)(l3:'a list).
  (APP(APP l1 l2) l3) = (APP l1 (APP l2 l3))' '
Induct_on 'l1' THEN
ASM_REWRITE_TAC [APP_def]
===== end here ===== *)
```

2. Apply the tactic `ASM_REWRITE_TAC [APP_def]` to both subgoals by highlighting it and doing **M-h e** or the expand function `e`.
3. The result is shown below. Notice, that the first application of `ASM_REWRITE_TAC [APP_def]` proves the base case; the second application of `ASM_REWRITE_TAC [APP_def]` proves the inductive step. As the same tactic is applied to both subgoals produced by `Induct_on 'l1'`, use the tactical `THEN` after `Induct_on 'l1'`.

```
- e(ASM_REWRITE_TAC [APP_def]);
OK..

Goal proved.
[] |- !l2 l3. APP (APP [] l2) l3 = APP [] (APP l2 l3)

Remaining subgoals:
> val it =

!h l2 l3. APP (APP (h::l1) l2) l3 = APP (h::l1) (APP l2 l3)
-----
!l2 l3. APP (APP l1 l2) l3 = APP l1 (APP l2 l3)
: proof
- e(ASM_REWRITE_TAC [APP_def]);
OK..

Goal proved.
[!l2 l3. APP (APP l1 l2) l3 = APP l1 (APP l2 l3)]
|- !h l2 l3. APP (APP (h::l1) l2) l3 = APP (h::l1) (APP l2 l3)
> val it =
Initial goal proved.
[] |- !l1 l2 l3. APP (APP l1 l2) l3 = APP l1 (APP l2 l3)
: proof
```

Now that we know our proof works, we package it up into a compound tactic, do the proof in one step using `TAC_PROOF`, and save the theorem under `APP_ASSOC`. Notice that now the proof is outside the multi-line comment and is visible to `Holmake`.

Do the following.

1. Add the following code to `exTypeScript.sml`.

```
val APP_ASSOC =
TAC_PROOF(([],
  '!(l1:'a list)(l2 : 'a list)(l3:'a list).
  (APP(APP l1 l2) l3) = (APP l1 (APP l2 l3))' ',
Induct_on 'l1' THEN
ASM_REWRITE_TAC [APP_def])

val _ = save_thm("APP_ASSOC", APP_ASSOC)
```

2. As usual, execute `Holmake` to rebuild the theory and export it.



## 11.5 Defining New Types and Their Properties in HOL

In this section, we show how to add new types to HOL and prove useful properties about them using HOL's built-in functions. Our description is only an introduction. Section 4.1 in *The HOL System DESCRIPTION* [HOL, 2015a] provides a thorough description of type definition with numerous examples.

### Example 11.5

In this example, we introduce a language of boolean formulas defined as the HOL type *bexp* and prove properties of *bexp*.

Do the following.

1. Create a new file *bexpScript.sml* with the following code.

```
structure bexpScript = struct
open HolKernel Parse boolLib bossLib;
open TypeBase boolTheory

val _ = new_theory "bexp";

val _ = export_theory ();
val _ = print_theory "-";

end
```

2. As usual, run [Holmake](#) in a terminal in the same subdirectory as *bexpScript.sml* to make sure the basic theory structure is correct.
3. Start an interactive HOL session within Emacs.
4. Load and open the following theories: *TypeBase*, *boolTheory*, and *bexpTheory* in HOL.
5. Add the following source code to *bexpScript.sml* immediately after

```
val _ = new_theory "bexp";
```

```
val _ = new_theory "bexp";

(*****)
(* Introduce the syntax of boolean expression bexp *)
(*****)
val _ = Datatype
`bexp = TT | FF | Not bexp | And bexp bexp | Or bexp bexp`;
```

6. Executing the above in the HOL interpreter produces the following result.

```
- val _ = Datatype
`bexp = TT | FF | Not bexp | And bexp bexp | Or bexp bexp`;
<<HOL message: Defined type: "bexp">>
```

8

*TypeBase* in HOL has several functions that prove properties of types. We illustrate their application on *bexp*.

Do the following.

1. Add the following code to *bexpScript.sml* after the [Datatype](#) definition.

```
(*****
(* Prove that identical bexps have identical components *)
*****)
val bexp_one_one = one_one_of `':bexp`
val _ = save_thm("bexp_one_one", bexp_one_one)

(*****
(* Prove that the different forms of bexp expressions are distinct *)
*****)
val bexp_distinct_clauses = distinct_of `':bexp`
val _ = save_thm("bexp_distinct_clauses", bexp_distinct_clauses)
```

2. Executing the above in HOL produces the following theorems as shown below. If there are no errors, you should execute [Holmake](#) within a terminal in the same subdirectory as *bexpScript.sml* to make sure the theory builds with no errors. The theorem *bexp\_one\_one* states that identical *bexp* formulas have the same constructor, i.e., *Not*, *And*, or *Or*, applied to the same arguments. The theorem *bexp\_distinct\_clauses* states that each form of *bexp* formulas is different from all the others.

```
- > val bexp_one_one =
  |- (!a a'. (Not a = Not a') <=> (a = a')) /\
    (!a0 a1 a0' a1'.
      (And a0 a1 = And a0' a1') <=> (a0 = a0') /\ (a1 = a1')) /\
      !a0 a1 a0' a1'. (Or a0 a1 = Or a0' a1') <=> (a0 = a0') /\ (a1 = a1'))
  : thm
- > val bexp_distinct_clauses =
  |- TT <> FF /\ (!a. TT <> Not a) /\ (!a1 a0. TT <> And a0 a1) /\
    (!a1 a0. TT <> Or a0 a1) /\ (!a. FF <> Not a) /\
    (!a1 a0. FF <> And a0 a1) /\ (!a1 a0. FF <> Or a0 a1) /\
    (!a1 a0 a. Not a <> And a0 a1) /\ (!a1 a0 a. Not a <> Or a0 a1) /\
    !a1' a1 a0' a0. And a0 a1 <> Or a0' a1'
```

◇

### Example 11.6

In this example, we add the semantics of *bexp* expressions by defining the function *bexpVal*. Do the following.

1. Open *bexpScript.sml* and start a HOL session. Make sure you have executed [Holmake](#) to rebuild the theory to include all the previous additions.
2. Add the following code defining *bexpVal*. Add it just before `val _ = export_theory();`

```
(*****
(* Define the semantics of bexp expressions *)
*****)
val bexpVal_def =
Define
  '(bexpVal TT = T) /\
  (bexpVal FF = F) /\
  (bexpVal (Not f) = ~(bexpVal f)) /\
  (bexpVal (And f1 f2) = ((bexpVal f1) /\ (bexpVal f2))) /\
  (bexpVal (Or f1 f2) = ((bexpVal f1) \/ (bexpVal f2)))'
```



3. The above code defines the boolean value for each of the five forms *bexp* formulas can have. There is one defining clause for each form. Notice that the definition of *bexpVal* is recursive. For example, the boolean value of `And f1 f2` is the conjunction of the values `bexpVal f1` and `bexpVal f2`.
4. Executing the above in HOL gives the following result.

```

- val bexpVal_def =
Define
  `(bexpVal TT = T) /\
   (bexpVal FF = F) /\
   (bexpVal (Not f) = ~(bexpVal f)) /\
   (bexpVal (And f1 f2) = ((bexpVal f1) /\ (bexpVal f2))) /\
   (bexpVal (Or f1 f2) = ((bexpVal f1) \/ (bexpVal f2)))`;
<<HOL message: more than one resolution of overloading was possible>>
Definition has been stored under "bexpVal_def"
> val bexpVal_def =
  |- (bexpVal TT <=> T) /\ (bexpVal FF <=> F) /\
    (!f. bexpVal (Not f) <=> ~bexpVal f) /\
    (!f1 f2. bexpVal (And f1 f2) <=> bexpVal f1 /\ bexpVal f2) /\
    (!f1 f2. bexpVal (Or f1 f2) <=> bexpVal f1 \/ bexpVal f2)
  : thm

```

10

Note that the defining theorem for *bexpVal* is named (by convention) *bexpVal\_def*. If everything works as shown above, execute `Holmake` to rebuild the theory.  $\diamond$

The following sequence of examples prove properties of the semantics of *bexp* expressions. These properties correspond to the properties of the underlying arithmetic operators in *arithmeticTheory*.

### Example 11.7

In this example we prove the double negation property. Do the following.

1. Open *bexpScript.sml* and start a HOL session. Make sure you have executed `Holmake` to rebuild the theory to include all the previous additions.
2. Add the following code defining *bexpVal*. Add it just before `val _ = export_theory();`

```

(*****
(* Prove bexpVal(Not(Not f)) = bexpVal(f)
*****
(* ===== start here =====
set_goal([], '!(f:bexp). bexpVal(Not(Not f)) = bexpVal(f) ' '
Induct_on 'f' THEN
ASM_REWRITE_TAC[bexpVal_def]
===== end here ===== *)

```

3. Execute `set_goal` and then apply the `Induct_on 'f'` tactic. The result is shown below.

11

```

- > val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !f. bexpVal (Not (Not f)) <=> bexpVal f

    : proofs
- OK..
5 subgoals:
> val it =

  bexpVal (Not (Not (Or f f'))) <=> bexpVal (Or f f')
  -----
  0. bexpVal (Not (Not f)) <=> bexpVal f
  1. bexpVal (Not (Not f')) <=> bexpVal f'

  bexpVal (Not (Not (And f f'))) <=> bexpVal (And f f')
  -----
  0. bexpVal (Not (Not f)) <=> bexpVal f
  1. bexpVal (Not (Not f')) <=> bexpVal f'

  bexpVal (Not (Not (Not f))) <=> bexpVal (Not f)
  -----
  bexpVal (Not (Not f)) <=> bexpVal f

  bexpVal (Not (Not FF)) <=> bexpVal FF

  bexpVal (Not (Not TT)) <=> bexpVal TT

  5 subgoals
  : proof

```

The five subgoals correspond to the five forms a *bexp* expression may have. The first two, **TT** and **FF** are non-recursive formulas. The remaining three are recursively defined. Hence, they each have an inductive hypothesis.

Rewriting using the definition *bexpVal\_def* and the inductive hypothesis, when available, will prove each of the subgoals. Backing up to the initial goal and applying induction followed by rewriting, proves the top-level goal. This is shown below.

12

```

- > val it =
  Initial goal:

    !f. bexpVal (Not (Not f)) <=> bexpVal f

    : proof
- e(Induct_on`f` THEN
  ASM_REWRITE_TAC[bexpVal_def]);
OK..
> val it =
  Initial goal proved.
|- !f. bexpVal (Not (Not f)) <=> bexpVal f
  : proof

```

As before, we use **TAC\_PROOF** to prove the theorem in one step.

1. Add to *bexpScript.sml* the following code.

```
val And_TT_thm =
TAC.PROOF([], '!(f:bexp). bexpVal(And TT f) = bexpVal(f) ' '),
Induct_on 'f' THEN
ASM.REWRITE_TAC[ bexpVal_def ])

val _ = save_thm("And_TT_thm", And_TT_thm)
```

2. Rebuild the theory and export it by executing [Holmake](#) in a terminal in the same subdirectory as *bexpScript.sml*. ◇

### Example 11.8

In this example, we prove that *And* is symmetric with respect to *bexpVal*. Do the following.

1. Open *bexpScript.sml* and start a HOL session. Make sure you have executed [Holmake](#) to rebuild the theory to include all the previous additions.
2. Add the following code defining *bexpVal*. Add it just before `val _ = export_theory();`

```
(*****
(* Prove bexpVal(And f1 f2) = bexpVal(And f2 f1) *)
(* *****)
(* ==== start here ====
set_goal([], 'f1 f2. bexpVal(And f1 f2) = bexpVal(And f2 f1) ' ')
Induct_on 'f1' THEN
PROVE_TAC[ bexpVal_def ]
===== end here ===== *)
```

3. Notice that this time we use [PROVE\\_TAC](#) instead of [ASM\\_REWRITE\\_TAC](#). This is because the inductive hypothesis in the recursive formulas are symmetrical and cause the subgoal to be endlessly rewritten. We use [PROVE\\_TAC](#) instead as it is sophisticated enough to avoid infinite rewrites of this kind.
4. To see this, execute `set_goal` followed by `Induct_on 'f1'`. The result is shown below.

13

```

- > val it =
  Initial goal:

    !f1 f2. bexpVal (And f1 f2) <=> bexpVal (And f2 f1)

    : proof
- e(Induct_on`f1`);
OK..
5 subgoals:
> val it =

    !f2. bexpVal (And (Or f1 f1') f2) <=> bexpVal (And f2 (Or f1 f1'))
-----
    0. !f2. bexpVal (And f1 f2) <=> bexpVal (And f2 f1)
    1. !f2. bexpVal (And f1' f2) <=> bexpVal (And f2 f1')

    !f2. bexpVal (And (And f1 f1') f2) <=> bexpVal (And f2 (And f1 f1'))
-----
    0. !f2. bexpVal (And f1 f2) <=> bexpVal (And f2 f1)
    1. !f2. bexpVal (And f1' f2) <=> bexpVal (And f2 f1')

    !f2. bexpVal (And (Not f1) f2) <=> bexpVal (And f2 (Not f1))
-----
    !f2. bexpVal (And f1 f2) <=> bexpVal (And f2 f1)

    !f2. bexpVal (And FF f2) <=> bexpVal (And f2 FF)

    !f2. bexpVal (And TT f2) <=> bexpVal (And f2 TT)

5 subgoals
: proof

```

The first two subgoals are provable using [ASM\\_REWRITE\\_TAC](#). However, the remaining recursive goals have assumptions that could cause their subgoals to be rewritten endlessly. If we use [PROVE\\_TAC\[bexpVal\\_def\]](#), all the subgoals are proved. This is shown below.

14

```

- > val it =
  Initial goal:

    !f1 f2. bexpVal (And f1 f2) <=> bexpVal (And f2 f1)

    : proof
- e(Induct_on`f1` THEN
PROVE_TAC[bexpVal_def]);
OK..
Meson search level: .....
Meson search level: .....
Meson search level: .....
Meson search level: .....
Meson search level: .....
> val it =
  Initial goal proved.
|- !f1 f2. bexpVal (And f1 f2) <=> bexpVal (And f2 f1)
: proof

```

As usual, we use `TAC_PROOF` to prove the theorem in one step and add it to `bexpScript.sml` using `save_thm`. Do the following.

1. Add to `bexpScript.sml` the following code.

```
val And_sym_thm =
TAC.PROOF([[], '!' f1 f2.bexpVal(And f1 f2) = bexpVal(And f2 f1) ' '),
Induct_on 'f1' THEN
PROVE_TAC[bexpVal_def])

val _ = save_thm("And_sym_thm", And_sym_thm)
```

2. Rebuild the theory and export it by executing `Holmake` in a terminal in the same subdirectory as `bexpScript.sml`. ◇

## 11.6 Exercises

**Exercise 11.6.1** Add to the theory `exTypeTheory` by proving the following theorem. Save it within `exTypeTheory`.

`[LENGTH_APP]`

$$\vdash \forall l_1 l_2. \text{LENGTH } (\text{APP } l_1 l_2) = \text{LENGTH } l_1 + \text{LENGTH } l_2$$

**Hint:** You will need to use `arithmeticTheory`. Pay attention to the theorem `ADD_CLAUSES`. Modify your `Script` file to include `arithmeticTheory`.

**Exercise 11.6.2** Add to the theory `exTypeTheory` by defining the function `Map` (note that “M” is capitalized and “ap” is lowercase). The behavior of `Map` is illustrated by

$$\text{Map } f [1;2;3;4] = [f 1; f 2; f 3; f 4]$$

Prove the following theorem. Save it within `exTypeTheory`.

`Map_APP`

$$\vdash \text{Map } f (\text{APP } l_1 l_2) = \text{APP } (\text{Map } f l_1) (\text{Map } f l_2)$$

**Exercise 11.6.3** Define a new theory `nexpTheory` corresponding to natural number expressions, which is similar to the example `bexpTheory` defined earlier on boolean expressions. Your implementation of `nexpTheory` should have the following datatypes, definitions, and theorems.

1. The datatype `nexp`:

$$\text{nexp} = \text{Num num} \mid \text{Add nexp nexp} \mid \text{Sub nexp nexp} \mid \text{Mult nexp nexp}$$

2. A definition of the **semantics** of `nexp` expressions. Call this semantic function `nexpVal`. Remember, by convention, the name of the defining theorem is `nexpVal.def`. Make sure you follow this convention to avoid confusing `HOL`.
3. Prove and save the following theorems as part of your version of `nexpTheory`. **Hint:** you will use `arithmeticTheory`. Pay close attention to the theorems in `arithmeticTheory`. Each of your theorems has a corresponding theorem in `arithmeticTheory` that you will likely need to use.

[Add\_0]

$$\vdash \forall f. \text{nexpVal } (\text{Add } (\text{Num } 0) f) = \text{nexpVal } f$$

[Add\_SYM]

$$\vdash \forall f_1 f_2. \text{nexpVal } (\text{Add } f_1 f_2) = \text{nexpVal } (\text{Add } f_2 f_1)$$

[Sub\_0]

$$\begin{aligned} \vdash \forall f. \\ & (\text{nexpVal } (\text{Sub } (\text{Num } 0) f) = 0) \wedge \\ & (\text{nexpVal } (\text{Sub } f (\text{Num } 0)) = \text{nexpVal } f) \end{aligned}$$

[Mult\_ASSOC]

$$\begin{aligned} \vdash \forall f_1 f_2 f_3. \\ & \text{nexpVal } (\text{Mult } f_1 (\text{Mult } f_2 f_3)) = \\ & \text{nexpVal } (\text{Mult } (\text{Mult } f_1 f_2) f_3) \end{aligned}$$

# Inductive Relations in HOL

---

In this chapter we introduce *inductive relations* as a means to describe transition systems. Defining relations inductively is one type of recursive definition. For example, we define the set of even numbers **even** with the following rules:

1. 0 is in **even**.
2. If  $n$  is in **even** then  $n + 2$  is in **even**.
3. **even** is the smallest set satisfying rules (1) and (2).

The third rule is essential to inductive definitions, i.e., if a set or relation is inductively defined, then it is the *smallest* set or relation that satisfies the rules. The rules make *positive* assertions about membership in a set or relation, i.e., statements about elements that *belong* to the set or relation, as opposed to elements that are outside the set or relation. In our definition of **even**, the first two rules are positive statements about membership. The set of numbers we think of as even numbers is the set  $\{0, 2, 4, 6, \dots\}$ . This set satisfies all three rules. To see this, we start with 0, which satisfies the first rule. From there, by repeated application of the second rule, we get  $\{0, 0 + 2, (0 + 2) + 2, ((0 + 2) + 2), \dots\} = \{0, 2, 4, 6, \dots\}$ . This set also satisfies the third rule as it is the smallest set of numbers satisfying the first two rules.

If the third rule were relaxed, then numbers we typically think of as *odd* satisfy the first two rules defining **even**. For example, the set  $\{0, 2, 4, 5, 6, 7, 8, 9, 10, 11, \dots\}$  satisfies the first two rules, but not the third. The numbers 7, 9, 11, etc. must be included if 5 is included. Specifically, if 5 is in **even**, then so is  $5 + 2$ . If  $5 + 2$  is in **even**, then so is  $(5 + 2) + 2$ , etc. What the third rule does is remove all extraneous members of the set defining **even** except for those elements common to *all* sets satisfying the rules.

In the remainder of this chapter, we will show how define and use relations inductively in HOL. In the chapters that follow, we will apply inductively-defined relations to transition systems in general, and state machines in particular.

## 12.1 Inductive Definitions in HOL

As an introduction to inductive definitions, we define *even* and *odd* inductively.

### Example 12.1

Do the following:

1. Create the file *hlsmlScript.sml* and add to it what is shown below in the shadow box.

```

(*****)
(* hlsm1Script.sml *)
(* Author: <your name> *)
(* Date: <today's date> *)
(*****)
structure hlsm1Script = struct

(* ==== Interactive mode ====
app load ["hlsm1Theory", "numLib", "arithmeticTheory"];
open hlsm1Theory numLib arithmeticTheory;
==== end interactive mode ==== *)

open HolKernel boolLib Parse bossLib
open numLib arithmeticTheory

(*****)
(* create a new theory *)
(*****)
val _ = new_theory "hlsm1";

val _ = export_theory ();
val _ = print_theory "-";

end (* structure *)

```

2. When you execute Holmake inside a terminal in the subdirectory containing `hlsm1Script.sml` you should see what is in the numbered session box below. This just checks to make sure that the script file works with Holmake prior to any definitions or proofs.

1

```

$ Holmake
/usr/local/share/HOL/bin/Holmake: Analysing hlsm1Script.sml
/usr/local/share/HOL/bin/Holmake: Trying to create directory .HOLMK for dependency files
Compiling hlsm1Script.sml
Linking hlsm1Script.uo to produce theory-builder executable
<<HOL message: Created theory "hlsm1">>
Exporting theory "hlsm1" ... done.
Theory "hlsm1" took 0.000s to build
Theory: hlsm1

Parents:
  list

/usr/local/share/HOL/bin/Holmake: Analysing hlsm1Theory.sml
/usr/local/share/HOL/bin/Holmake: Analysing hlsm1Theory.sig
Compiling hlsm1Theory.sig
Compiling hlsm1Theory.sml

```

Now that we know our script file works with Holmake, we add the definitions of *even* and *odd*.

1. Add the following to the file `hlsm1Script.sml` as shown below in the shadow box. **Insert the definitions after `new_theory "hlsm1"` and before `export_theory ()`.**



```

val (even_rules, even_induction, even_cases) =
Hol_reln
`even 0 /\
  (!n. even n ==> even (n + 2))`;

val _ = save_thm("even_rules",even_rules);
val _ = save_thm("even_induction",even_induction);
val _ = save_thm("even_cases",even_cases);

val (odd_rules, odd_induction, odd_cases) =
Hol_reln
`odd 1 /\
  (!n. odd n ==> odd (n + 2))`;

val _ = save_thm("odd_rules",odd_rules);
val _ = save_thm("odd_induction",odd_induction);
val _ = save_thm("odd_cases",odd_cases);

```

2. Open a HOL session within Emacs, and execute the above ML code within HOL. What you should see is shown below in the numbered session box.

2

```

- val (even_rules, even_induction, even_cases) =
Hol_reln
`even 0 /\
  (!n. even n ==> even (n + 2))`;
> val even_rules =
  |- even 0 /\ !n. even n ==> even (n + 2)
  : thm
val even_induction =
  |- !even'.
    even' 0 /\ (!n. even' n ==> even' (n + 2)) ==>
    !a0. even a0 ==> even' a0
  : thm
val even_cases =
  |- !a0. even a0 <=> (a0 = 0) /\ ?n. (a0 = n + 2) /\ even n
  : thm
- val _ = save_thm("even_rules",even_rules);
val _ = save_thm("even_induction",even_induction);
val _ = save_thm("even_cases",even_cases);
- - - val (odd_rules, odd_induction, odd_cases) =
Hol_reln
`odd 1 /\
  (!n. odd n ==> odd (n + 2))`;
> val odd_rules =
  |- odd 1 /\ !n. odd n ==> odd (n + 2)
  : thm
val odd_induction =
  |- !odd'.
    odd' 1 /\ (!n. odd' n ==> odd' (n + 2)) ==> !a0. odd a0 ==> odd' a0
  : thm
val odd_cases =
  |- !a0. odd a0 <=> (a0 = 1) /\ ?n. (a0 = n + 2) /\ odd n
  : thm
- val _ = save_thm("odd_rules",odd_rules);
val _ = save_thm("odd_induction",odd_induction);
val _ = save_thm("odd_cases",odd_cases);
- - -

```

As normal, whenever we add definitions or proofs to a script file, we save the additions and execute Holmake to make sure all is in order. The following numbered session box is what you should see after executing Holmake in a terminal open to the subdirectory containing *hlsm1Script.sml*. To save space, we have omitted much of the output.

3

```

$ Holmake
/usr/local/share/HOL/bin/Holmake: Analysing hlsm1Script.sml
Compiling hlsm1Script.sml
Linking hlsm1Script.uo to produce theory-builder executable
<<HOL message: Created theory "hlsm1">>
Saved theorem _____ "even_rules"

... output omitted ...

Saved theorem _____ "odd_cases"
Exporting theory "hlsm1" ... done.
Theory "hlsm1" took 0.832s to build
Theory: hlsm1

Parents:
  list

Term constants:
  even   :num -> bool
  odd    :num -> bool

Definitions:

... output omitted ...

Theorems:

... output omitted ...

/usr/local/share/HOL/bin/Holmake: Analysing hlsm1Theory.sig
Compiling hlsm1Theory.sig
/usr/local/share/HOL/bin/Holmake: Analysing hlsm1Theory.sml
Compiling hlsm1Theory.sml

```

Look at the HOL function [Hol\\_reln](#) in the following code snippet.

```

val (even_rules, even_induction, even_cases) =
  Hol_reln
  'even 0 /\
  (!n. even n ==> even (n + 2))';

```

The HOL code above produces three theorems shown below.

[\[even\\_rules\]](#)

$$\vdash \text{even } 0 \wedge \forall n. \text{even } n \Rightarrow \text{even } (n + 2)$$

[\[even\\_induction\]](#)

$$\begin{aligned} &\vdash \forall \text{even}'. \\ &\quad \text{even}' 0 \wedge (\forall n. \text{even}' n \Rightarrow \text{even}' (n + 2)) \Rightarrow \\ &\quad \forall a_0. \text{even } a_0 \Rightarrow \text{even}' a_0 \end{aligned}$$

[\[even\\_cases\]](#)

$$\vdash \forall a_0. \text{even } a_0 \iff (a_0 = 0) \vee \exists n. (a_0 = n + 2) \wedge \text{even } n$$

The first theorem *even\_rules* is a commonly used description of even numbers: 0 is even, and if  $n$  is even then so is  $n+2$ . The second theorem *even\_induction* is an induction principle using the fact that the inductive definition of *even* is the *smallest set of numbers satisfying the even\_rules*. In other words, if a relation *even'* satisfies the same rules as *even*, then when *even* is true *even'* must be true, too. Finally, the third theorem *even\_cases* states that if  $a_0$  is even, then  $a_0$  is either 0 or there is an even number  $n$  such that  $a_0 = n + 2$ .  $\diamond$

**Example 12.2**

In this example, we define *odd* numbers in an analogous way to *even* numbers. Do the following:

1. Add to the file *hlsmlScript.sml* what is shown below in the shadow box after the code defining *even* and saving the three theorems characterizing *even*.

```
val (odd_rules, odd_induction, odd_cases) =
  Hol_reln
  `odd 1 /\
    (!n. odd n ==> odd (n + 2)) `;

val _ = save_thm("odd_rules", odd_rules);
val _ = save_thm("odd_induction", odd_induction);
val _ = save_thm("odd_cases", odd_cases);
```

2. Open a HOL session within Emacs, and execute the code within HOL. What you should see is shown below in the numbered session box.

4

```
- val (odd_rules, odd_induction, odd_cases) =
  Hol_reln
  `odd 1 /\
    (!n. odd n ==> odd (n + 2)) `;

val _ = save_thm("odd_rules", odd_rules);
val _ = save_thm("odd_induction", odd_induction);
val _ = save_thm("odd_cases", odd_cases);
> val odd_rules =
  |- odd 1  n. odd n  odd (n + 2)
  : thm
val odd_induction =
  |- odd'. odd' 1  (n. odd' n  odd' (n + 2))  a0. odd a0  odd' a0
  : thm
val odd_cases =
  |- a0. odd a0  (a0 = 1)  n. (a0 = n + 2)  odd n
  : thm
```

3. As usual, save everything you have added to *hlsmlScript.sml* and in a terminal open to the subdirectory containing *hlsmlScript.sml*, execute `Holmake cleanAll` followed by `Holmake` to ensure *hlsmlScript.sml* compiles correctly.

The results are similar to the definition of *even*. We have the following three theorems characterizing the relation *odd*.

[odd\_rules]

$$\vdash \text{odd } 1 \wedge \forall n. \text{odd } n \Rightarrow \text{odd } (n + 2)$$

[odd\_induction]

$$\begin{aligned} &\vdash \forall \text{odd}'. \\ &\quad \text{odd}' 1 \wedge (\forall n. \text{odd}' n \Rightarrow \text{odd}' (n + 2)) \Rightarrow \\ &\quad \forall a_0. \text{odd } a_0 \Rightarrow \text{odd}' a_0 \end{aligned}$$

[odd\_cases]

$$\vdash \forall a_0. \text{odd } a_0 \iff (a_0 = 1) \vee \exists n. (a_0 = n + 2) \wedge \text{odd } n$$

Similar to the definition of *even*, we have three theorems: (1) *odd\_rules* defining odd numbers in a typical way, (2) the induction theorem resulting from the fact that *odd* is the smallest set of numbers satisfying the rules, and (3) the *odd\_cases* theorem saying that all odd numbers  $a_0$  are either 1 or there is an  $n$  such that  $a_0 = n + 2$ . ◇

**Example 12.3**

In this example, we define *Even* and *Odd* numbers in a mutually inductive fashion. Do the following:

1. Add to the file *hlsmlScript.sml* what is shown below in the shadow box after the code defining and characterizing *even* and *odd*.

```
val (Even_Odd_rules, Even_Odd_induction, Even_Odd_cases) =
  Hol_reln
  `Even 0 /\
    (!n. Odd n ==> Even (n + 1)) /\
    (!n. Even n ==> Odd (n + 1)) `;

val _ = save_thm("Even_Odd_rules", Even_Odd_rules);
val _ = save_thm("Even_Odd_induction", Even_Odd_induction);
val _ = save_thm("Even_Odd_cases", Even_Odd_cases);
```

2. Open a HOL session within Emacs, and execute the code within HOL. What you should see is shown below in the numbered session box.

```
- val (Even_Odd_rules, Even_Odd_induction, Even_Odd_cases) =
Hol_reln
`Even 0 /\
  (!n. Odd n ==> Even (n + 1)) /\
  (!n. Even n ==> Odd (n + 1)) `;

val _ = save_thm("Even_Odd_rules", Even_Odd_rules);
val _ = save_thm("Even_Odd_induction", Even_Odd_induction);
val _ = save_thm("Even_Odd_cases", Even_Odd_cases);
> val Even_Odd_rules =
  |- Even 0 (n. Odd n => Even (n + 1)) n. Even n => Odd (n + 1)
  : thm
val Even_Odd_induction =
  |- Even' Odd'.
    Even' 0 (n. Odd' n => Even' (n + 1))
    (n. Even' n => Odd' (n + 1))
    (a0. Even a0 => Even' a0) a1. Odd a1 => Odd' a1
  : thm
val Even_Odd_cases =
  |- (a0. Even a0 (a0 = 0) n. (a0 = n + 1) => Odd n)
    a1. Odd a1 n. (a1 = n + 1) => Even n
  : thm
```

5

3. As usual, save everything you have added to *hlsmlScript.sml* and in a terminal open to the subdirectory containing *hlsmlScript.sml*, execute `Holmake cleanAll` followed by `Holmake` to ensure *hlsmlScript.sml* compiles correctly.

The results are similar to the definition of *even*. We have the following three theorems characterizing the relation *odd*.

[Even\_Odd\_rules]

$$\vdash \text{Even } 0 \wedge (\forall n. \text{Odd } n \Rightarrow \text{Even } (n + 1)) \wedge \\ \forall n. \text{Even } n \Rightarrow \text{Odd } (n + 1)$$

[Even\_Odd\_induction]

$$\vdash \forall \text{Even}' \text{ Odd}'. \\ \text{Even}' 0 \wedge (\forall n. \text{Odd}' n \Rightarrow \text{Even}' (n + 1)) \wedge \\ (\forall n. \text{Even}' n \Rightarrow \text{Odd}' (n + 1)) \Rightarrow \\ (\forall a_0. \text{Even } a_0 \Rightarrow \text{Even}' a_0) \wedge \forall a_1. \text{Odd } a_1 \Rightarrow \text{Odd}' a_1$$

[Even\_Odd\_cases]

$$\begin{aligned} \vdash (\forall a_0. \text{Even } a_0 &\iff (a_0 = 0) \vee \exists n. (a_0 = n + 1) \wedge \text{Odd } n) \wedge \\ &\forall a_1. \text{Odd } a_1 \iff \exists n. (a_1 = n + 1) \wedge \text{Even } n \end{aligned}$$

Notice that both *Even* and *Odd* are defined using the other. This is what makes the definitions mutually inductive. The base cases are the same as before: 0 is *Even* and 1 is *Odd*. The mutually inductive definitions are what we expect: if  $n$  is *Even* then  $n + 1$  is *Odd*, and if  $n$  is *Odd* then  $n + 1$  is *Even*. The associated induction rule again reflects the fact that both *Even* and *Odd* are defined as the smallest sets satisfying the rules. The associated cases rule is a reflection of *Even\_Odd\_rules*.  $\diamond$

## 12.2 Reasoning About Inductive Relations

At this point, we have defined the inductive relations *even*, *odd*, *Even*, and *Odd*. Add to these definitions, the built-in definitions *EVEN* and *ODD* in HOL, which are recursively defined over natural numbers  $n$ .

```
[EVEN]
⊢ (EVEN 0 ⟷ T) ∧ ∀n. EVEN (SUC n) ⟷ ¬EVEN n

[ODD]
⊢ (ODD 0 ⟷ F) ∧ ∀n. ODD (SUC n) ⟷ ¬ODD n
```

With the above, that makes *three* definitions of odd and even numbers. A natural question is, *are they equivalent definitions?* In this section, and in the exercises at the end of this chapter, we prove that the answer in all cases is yes, they are logically equivalent relations.

In the next series of examples, we prove properties about *even*, *Even*, *EVEN*, *odd*, *Odd*, and *ODD* with the goal of showing their equivalence. **Before doing the following examples, make sure you exit from the HOL sessions used to define *even*, *Even*, *odd* and *Odd* and run Holmake to ensure your HOL theory files are up to date.**

Our proofs depend on the HOL built-in theory *arithmeticTheory* and the HOL built-in library *numLib*. Details on *arithmeticTheory* and *numLib* are found in the HOL on-line documentation as well as in the HOL Reference Manual [HOL, 2015c].

To use the functions in *numLib*, we need to load and open it using `load "numLib"` followed by `open numLib`. In the proofs that follow, often we will use `ARITH_CONV`. `ARITH_CONV` is a decision procedure for arithmetic formulas on natural numbers. The next example shows how to use `ARITH_CONV` to generate useful arithmetic theorems.

### 12.2.1 Even\_rules

In this example, we prove that *Even* behaves the way *even* is defined. This is given by the theorem `Even_rules`.

```
[Even_rules]
⊢ Even 0 ∧ ∀n. Even n ⇒ Even (n + 2)
```

#### Example 12.4

Start a HOL window and do `load "numLib";` followed by `open numLib;`. Suppose we wish to show that  $(x + 5 = y + 2) = (y = x + 3)$ . We use `ARITH_CONV` to do all the hard work of proving the underlying theorem.

Do the following.

1. Open a HOL session, load and open *numLib*. Execute the following in the HOL session.

```
- ARITH_CONV``(x + 5 = y + 2) = (y = x + 3)``;
```

2. What you should see is in the numbered session box below.

```
- ARITH_CONV``(x + 5 = y + 2) = (y = x + 3)``;
> val it =
  |- ((x + 5 = y + 2) <=> (y = x + 3)) <=> T
  : thm
```

6

3. Notice the  $\text{<=> T}$  at the end of the theorem. Often, we are interested in replacing the left hand term  $(x + 5 = y + 2)$  with the right hand term  $(y = x + 3)$ , so we do not need the  $\text{<=> T}$ . To get rid of  $T$ , we simplify the theorem using `REWRITE_RULE []`. This is shown in the numbered session box below.

```
- REWRITE_RULE[] (ARITH_CONV``(x + 5 = y + 2) = (y = x + 3)``);
> val it =
  |- (x + 5 = y + 2) <=> (y = x + 3)
  : thm
```

7

Conveniently, HOL has several high-level decision procedures, such as `ARITH_CONV` and `PROVE_TAC`. These decision procedures often eliminate the need for intricate and laborious manipulations of expressions within theorems and/or goals.  $\diamond$

Armed with `ARITH_CONV`, we work through a series of proofs of properties of *even*, *Even*, *EVEN*, *odd*, *Odd*, and *ODD* that show their equivalence. These proofs are part of the examples that follow and the exercises at the end of the chapter.

### Example 12.5

Recall that the relation *Even* was mutually defined with *Odd*. Specifically,

[Even\_Odd\_rule]

$$\vdash \text{Even } 0 \wedge (\forall n. \text{Odd } n \Rightarrow \text{Even } (n + 1)) \wedge \\ \forall n. \text{Even } n \Rightarrow \text{Odd } (n + 1)$$

To relate the definitions of *Even* and *even*, we prove the theorem *Even\_rules*, whose form matches that of *even* in the *even\_rules* theorem.

[Even\_rules]

$$\vdash \text{Even } 0 \wedge \forall n. \text{Even } n \Rightarrow \text{Even } (n + 2)$$

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*.

```
app load ["hlsmlTheory", "numLib", "arithmeticTheory"];
open hlsmlTheory numLib arithmeticTheory;
```

2. We set the goal.

```
- set_goal([], `` (Even 0) /\ !n. (Even n) ==> (Even (n + 2)) ``);
```

The result is shown below.

```
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      Even 0 /\ !n. Even n ==> Even (n + 2)

: proofs
```

8

3. We note that the first conjunct in the goal `Even 0`, is a clause in the *Even\_rules* theorem. We rewrite the goal using *Even\_rules* and then simplify the remaining goal as much as possible using `STRIP_TAC`.

```
REWRITE_TAC[Even_Odd_rules] THEN
REPEAT STRIP_TAC
```

The result is shown below.

```
- OK..
1 subgoal:
> val it =

  Even (n + 2)
  -----
  Even n
  : proof
```

9

4. Next, we use the *Even\_Odd\_rules* coupled with the assumption `Even n` to introduce the assumption `Odd (n + 1)`. We do this using `IMP_RES_TAC Even_Odd_rules`.

```
IMP_RES_TAC Even_Odd_rules
```

The result is shown below.

```
- OK..
1 subgoal:
> val it =

  Even (n + 2)
  -----
  0. Even n
  1. Odd (n + 1)
  : proof
```

10

5. We use *Even\_Odd\_rules* again with `Odd (n + 1)` in the assumptions to add `Even (n + 1 + 1)` to the assumptions.

```
IMP_RES_TAC Even_Odd_rules
```

The result is shown below.

```
- OK..
1 subgoal:
> val it =
```

```
Even (n + 2)
-----
0. Even n
1. Odd (n + 1)
2. Even (n + 1 + 1)
: proof
```

11

6. Next, we use `ARITH_CONV ``n + 2 = n + 1 + 1``` within `PROVE_TAC` to finish the proof.

```
PROVE_TAC[ARITH_CONV ``n + 2 = n + 1 + 1``]
```

The result is shown below.

```
- OK..
Meson search level: .....
```

```
.... output omitted ....
```

```
> val it =
Initial goal proved.
|- Even 0 /\ !n. Even n ==> Even (n + 2)
: proof
```

12

7. Finally, we add the following code to `hlsmlScript.sml` and compile it using `Holmake` within a terminal.

```
val Even_rules =
TAC_PROOF([[] , `` (Even 0) /\ !n. (Even n) ==> (Even (n + 2)) ``),
REWRITE_TAC[Even_Odd_rules] THEN
REPEAT STRIP_TAC THEN
IMP_RES_TAC Even_Odd_rules THEN
IMP_RES_TAC Even_Odd_rules THEN
PROVE_TAC[ARITH_CONV ``n + 2 = n + 1 + 1``])

val _ = save_thm("Even_rules", Even_rules);
```

◇

### 12.2.2 Odd\_rules

The theorem `Odd_rules` corresponds to `Even_rules`. It shows that *Odd* behaves the same way *odd* is defined.

```
[Odd_rules]
⊢ Odd 1 ∧ ∀n. Odd n ⇒ Odd (n + 2)
```

The proof is part of Exercise 12.3.1.

### 12.2.3 not\_odd\_0

In this example, we prove that 0 is not *odd*.

```
[not_odd_0]
⊢ ¬odd 0
```



**Example 12.6**

This is one of the theorems that relates *even* and *odd* relations.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``~odd 0``);
```

The result is shown below.

```
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      ~odd 0

: proofs
```

13

2. We simplify the goal as much as possible using REPEAT STRIP\_TAC.

```
REPEAT STRIP_TAC
```

The result is shown below.

```
- OK..
1 subgoal:
> val it =

  F
  -----
    odd 0
: proof
```

14

3. One way to show that `odd 0` is false is to take advantage of the *odd\_cases* theorem. We do this by doing `IMP_RES_TAC odd_cases`.

```
IMP_RES_TAC odd_cases
```

The result is shown below.

```
- OK..
2 subgoals:
> val it =

  F
  -----
    0. odd 0
    1. 0 = n + 2
    2. odd n
    3. !a0. (a0 = 0 + 2) ==> odd a0

  F
  -----
    0. odd 0
    1. 0 = 1
    2. !a0. (a0 = 0 + 2) ==> odd a0

2 subgoals
: proof
```

15

4. We see that both subgoals have false assumptions,  $0 = 1$  and  $0 = n + 2$ . We use `ARITH_CONV` to show both are false and supply the theorems to `PROVE_TAC`.

```
PROVE_TAC[(ARITH_CONV``(0 = n + 2)``), (ARITH_CONV``(0 = 1)``)]
```

The result is shown below.

```
- OK..
Meson search level: ..

Goal proved.
[...] |- F

Remaining subgoals:
> val it =

F
-----
0.  odd 0
1.  0 = n + 2
2.  odd n
3.  !a0. (a0 = 0 + 2) ==> odd a0
: proof
```

16

We apply `PROVE_TAC` again to the remaining goal.

```
PROVE_TAC[(ARITH_CONV``(0 = n + 2)``), (ARITH_CONV``(0 = 1)``)]
```

The result is shown below.

```
- OK..
Meson search level: ..

Goal proved.
[....] |- F

Goal proved.
[.] |- F
> val it =
  Initial goal proved.
  |- ~odd 0
  : proof
```

17

5. Finally, we add the following code to *hlsmlScript.sml* and compile it with Holmake within a terminal.

```
val not_odd_0 =
  TAC_PROOF([], ``~odd 0``,
    REPEAT STRIP_TAC THEN
    IMP_RES_TAC odd_cases THEN
    PROVE_TAC[(ARITH_CONV``(0 = n + 2)``), (ARITH_CONV``(0 = 1)``)]

val _ = save_thm("not_odd_0", not_odd_0)
```

◇

## 12.2.4 not\_even\_1

The theorem `not_even_1` states that 1 is not even, i.e.,

```
[not_even_1]
```

$$\vdash \neg \text{even } 1$$

This theorem is proved in a similar way to *not\_odd\_0*. The proof is Exercise 12.3.2.

### 12.2.5 even\_not\_odd

The theorem *even\_not\_odd* states that if  $n$  is *even* then it is not *odd*.

```
[even_not_odd]

$$\vdash \forall n. \text{even } n \Rightarrow \neg \text{odd } n$$

```

First, we prove a useful lemma about not being odd, *odd\_not\_even\_lemma1*.

```
[odd_not_even_lemma1]

$$\vdash \neg \text{odd } 0 \wedge \forall n. \neg \text{odd } n \Rightarrow \neg \text{odd } (n + 2)$$

```

#### Example 12.7

The lemma states that not being *odd* follows the same pattern as being *even* or *odd*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``~odd 0 /\ !n. ~(odd n) ==> ~(odd (n + 2))``);
```

The result is shown below.

```
> val it =
Proof manager status: 1 proof.
1. Incomplete goalstack:
  Initial goal:

    ~odd 0 /\ !n. ~odd n ==> ~odd (n + 2)

: proofs
```

18

2. We note that we have already proved 0 is not odd as theorem *not\_odd\_0*. We rewrite the goal with the theorem.

```
REWRITE_TAC[not_odd_0]
```

The result is shown below.

```
- > val it = () : unit
- - - - - OK..
1 subgoal:
> val it =

  !n. ~odd n ==> ~odd (n + 2)

: proof
```

19

3. We simplify the goal as much as possible using `REPEAT STRIP_TAC`.

```
REPEAT STRIP_TAC
```

The result is shown below.

```
- OK..
1 subgoal:
> val it =
```

20

```
F
-----
0. ~odd n
1. odd (n + 2)
: proof
```

4. To set up a contradiction in the assumptions, we use *odd\_cases* to enrich the assumptions.

```
IMP_RES_TAC odd_cases
```

The result is shown below.

```
- OK..
2 subgoals:
> val it =
```

21

```
F
-----
0. ~odd n
1. odd (n + 2)
2. n + 2 = n' + 2
3. odd n'
4. !a0. (a0 = n + 2 + 2) ==> odd a0

F
-----
0. ~odd n
1. odd (n + 2)
2. n + 2 = 1
3. !a0. (a0 = n + 2 + 2) ==> odd a0

2 subgoals
: proof
```

5. We see in the assumptions of the first subgoal,  $n + 2 = 1$  is false. In the assumptions of the remaining subgoal,  $n + 2 = n' + 2$  implies  $n = n'$ , which makes  $\sim \text{odd } n$  and  $\text{odd } n'$  contradictory statements. We use *PROVE\_TAC* augmented with theorems stating  $\sim (n + 2 = 1)$  and  $(n + 2 = n' + 2) = (n = n')$  obtained using *ARITH\_CONV*.

```
PROVE_TAC
[ (ARITH_CONV `` ~ (n + 2 = 1) ``),
  (ARITH_CONV `` (n + 2 = n' + 2) = (n = n') ``) ]
```

The results are shown below.

```

- OK..
Meson search level: ..

Goal proved.
[....] |- F

Remaining subgoals:
> val it =

F
-----
0.  ~odd n
1.  odd (n + 2)
2.  n + 2 = n' + 2
3.  odd n'
4.  !a0. (a0 = n + 2 + 2) ==> odd a0
: proof

```

22

We apply `PROVE_TAC` again to the remaining goal

```

PROVE_TAC
[ (ARITH_CONV ``~(n+2 = 1) ``),
  (ARITH_CONV ``(n + 2 = n' + 2) = (n = n') ``) ]

```

The result is shown below.

```

- OK..
Meson search level: .....

Goal proved.
[.....] |- F
> val it =
  Initial goal proved.
  |- ~odd 0 /\ !n. ~odd n ==> ~odd (n + 2)
  : proof

```

23

6. Finally, we add the following code to *hsm1Script.sml* and compile it with `Holmake` within a terminal.

```

val odd_not_even_lemmal =
TAC_PROOF([[] , ``~odd 0 /\ !n. ~(odd n) ==> ~(odd (n + 2)) ``),
REWRITE_TAC[not_odd_0] THEN
REPEAT STRIP_TAC THEN
IMP_RES_TAC odd_cases THEN
PROVE_TAC
[ (ARITH_CONV ``~(n+2 = 1) ``),
  (ARITH_CONV ``(n + 2 = n' + 2) = (n = n') ``) ]

```

◇

### Example 12.8

We now prove *even\_not\_odd*.

1. We open a HOL session, load and open *hsm1Theory*, *numLib*, and *arithmeticTheory*. We set the goal.

```

set_goal([[] , ``!n. even n ==> ~odd n ``);

```

The result is shown below.

```

> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !n. even n ==> ~odd n

    : proofs

```

24

2. The form of our goal matches the form of **rule induction** for *even* given by *even\_induction*. We use `Induct_on `even``, which is smart enough to determine what kind of induction we want and if what we want is correct. Notice, the result differs from induction on the natural number  $n$ .

```
Induct_on `even`
```

The result is shown below.

```

- OK..
1 subgoal:
> val it =

  ~odd 0 /\ !n. even n /\ ~odd n ==> ~odd (n + 2)

  : proof

```

25

3. We see the above goal is covered by *odd\_not\_even\_lemma1*, which we just proved. We use `PROVE_TAC` augmented with *odd\_not\_even\_lemma1* to finish the proof.

```
PROVE_TAC[odd_not_even_lemma1]
```

The result is shown below.

```

- OK..
Meson search level: .....

Goal proved.
|- ~odd 0 /\ !n. even n /\ ~odd n ==> ~odd (n + 2)
> val it =
  Initial goal proved.
  |- !n. even n ==> ~odd n
  : proof

```

26

4. Finally, we add the following code to *hlsmlScript.sml* and compile it with `Holmake` within a terminal.

```

val even_not_odd =
  TAC_PROOF([[] , ``!n.even n ==> ~odd n``),
  Induct_on `even` THEN
  PROVE_TAC[odd_not_even_lemma1])

val _ = save_thm("even_not_odd", even_not_odd)

```

◇

### 12.2.6 odd\_not\_even

The theorem *odd\_not\_even* states that if  $n$  is odd then  $n$  cannot be even. The proof is Exercise 12.3.3.

[\[odd\\_not\\_even\]](#)

$\vdash \forall n. \text{odd } n \Rightarrow \neg \text{even } n$

### 12.2.7 even\_theorem

The theorem *even\_theorem* states that  $n$  is even if and only if  $n + 2$  is even.

```
[even_theorem]
⊢ ∀n. even n ⇔ even (n + 2)
```

The first part of the bi-conditional already is proved as part of *even\_rules*:

```
[even_rules]
⊢ even 0 ∧ ∀n. even n ⇒ even (n + 2)
```

Our first task is to prove the converse, *even\_lemma1*:

```
[even_lemma1]
⊢ ∀n. even (n + 2) ⇒ even n
```

#### Example 12.9

We prove *even\_lemma1*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``!n. (even (n+2) ==> even n)``);
```

The result is shown below.

```
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !n. even (n + 2) ==> even n

: proofs
```

27

2. We induct on the natural number  $n$  and use the fact that *even 0* is true as stated in *even\_rules*.

```
Induct_on `n` THEN
REWRITE_TAC[even_rules]
```

The result is shown below.

```
- OK..
1 subgoal:
> val it =

  even (SUC n + 2) ==> even (SUC n)
  -----
  even (n + 2) ==> even n
  : proof
```

28

3. We simplify the goal as much as possible using `REPEAT STRIP_TAC`.

```
REPEAT STRIP_TAC
```

The result is shown below.

```

- OK..
1 subgoal:
> val it =

  even (SUC n)
  -----
  0. even (n + 2) ==> even n
  1. even (SUC n + 2)
: proof

```

29

4. We enhance the assumptions with *even\_cases* through `IMP_RES_TAC`.

```
IMP_RES_TAC even_cases
```

The resulting two subgoals are shown below.

```

- OK..
2 subgoals:
> val it =

  even (SUC n)
  -----
  0. even (n + 2) ==> even n
  1. even (SUC n + 2)
  2. SUC n + 2 = n' + 2
  3. even n'
  4. !a0. (a0 = SUC n + 2 + 2) ==> even a0

  even (SUC n)
  -----
  0. even (n + 2) ==> even n
  1. even (SUC n + 2)
  2. SUC n + 2 = 0
  3. !a0. (a0 = SUC n + 2 + 2) ==> even a0

2 subgoals
: proof

```

30

5. Looking at the first subgoal and its assumptions, we see that  $\text{SUC } n + 2 = 0$  is false, which proves the first subgoal. Looking at the second subgoal, we see from  $\text{SUC } n + 2 = n' + 2$  that  $\text{SUC } n = n'$ . Thus, the subgoal `even (SUC n)` and the assumption `even n'` are the same, which proves the second subgoal. To handle both cases, we use `PROVE_TAC` augmented with the arithmetic relations shown below.

We apply `PROVE_TAC` to the first subgoal.

```

PROVE_TAC[ (ARITH_CONV``~(SUC n + 2 = 0)``),
            (ARITH_CONV``(SUC n + 2 = n' + 2) = (SUC n = n')``) ]

```

The result is shown below.



```

- OK..
Meson search level: ....

Goal proved.
[....] |- even (SUC n)

Remaining subgoals:
> val it =

    even (SUC n)
    -----
    0. even (n + 2) ==> even n
    1. even (SUC n + 2)
    2. SUC n + 2 = n' + 2
    3. even n'
    4. !a0. (a0 = SUC n + 2 + 2) ==> even a0
: proof

```

31

We apply `PROVE_TAC` to the remaining subgoal.

```

PROVE_TAC[(ARITH_CONV``(SUC n + 2 = 0)``),
           (ARITH_CONV``(SUC n + 2 = n' + 2) = (SUC n = n')``)]

```

The result is shown below.

```

- OK..
Meson search level: .....

Goal proved.
[.....] |- even (SUC n)

Goal proved.
[..] |- even (SUC n)

Goal proved.
[.] |- even (SUC n + 2) ==> even (SUC n)
> val it =
    Initial goal proved.
    |- !n. even (n + 2) ==> even n
: proof

```

32

6. Finally, we add the following code to *hlsm1Script.sml* and compile it with `Holmake` within a terminal.

```

val even_lemmal =
  TAC_PROOF([], ``!n. (even (n+2) ==> even n)``),
  Induct_on `n` THEN
  REWRITE_TAC[even_rules] THEN
  REPEAT STRIP_TAC THEN
  IMP_RES_TAC even_cases THEN
  PROVE_TAC[(ARITH_CONV``(SUC n + 2 = 0)``),
            (ARITH_CONV``(SUC n + 2 = n' + 2) = (SUC n = n')``)]

```

◇

With *even\_lemmal* and *even\_rules* we prove *even\_theorem* in one step.

### Example 12.10

1. Set the goal.

```

set_goal([], ``!n. even n = even (n + 2)``);

```

The result is shown below.

```
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
      Initial goal:

      !n. even n <=> even (n + 2)

  : proofs
```

33

2. Prove the goal in one step using `PROVE_TAC` augmented with *even\_lemma1* and *even\_rules*.

```
PROVE_TAC[even_lemma1,even_rules]
```

The result is shown below.

```
- OK..
Meson search level: .....
> val it =
  Initial goal proved.
  |- !n. even n <=> even (n + 2)
  : proof
```

34

3. Add the following code to *hlsmlScript.sml* and compile it with `Holmake` within a terminal.

```
val even_theorem =
  TAC_PROOF([], ``!n. even n = even (n + 2) ``),
  PROVE_TAC[even_lemma1,even_rules])

val _ = save_thm("even_theorem", even_theorem)
```

◇

### 12.2.8 odd\_theorem

The theorem *odd\_theorem* states that  $n$  is odd if and only if  $n + 2$  is odd. This proof is Exercise 12.3.4D.

### 12.2.9 not\_even\_lemma

The theorem *not\_even\_lemma* states that if  $n$  is not even then  $n + 1$  is even. Notice that this theorem relates properties of numbers  $n$  and  $n + 1$ , where *even* is defined in terms of  $n$  and  $n + 2$ .

[not\_even\_lemma]  
 $\vdash \neg \text{even } n \Rightarrow \text{even } (n + 1)$

#### Example 12.11

We prove *not\_even\_lemma*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``~even n ==> even (n+1) ``);
```

The result is shown below.

```

> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      ~even n ==> even (n + 1)

: proofs

```

35

2. We induct on natural number  $n$  and dispense with the subgoal corresponding to the base case  $n = 0$ , by rewriting both the base case and inductive step with *even\_rules*.

```

Induct_on `n` THEN
REWRITE_TAC[even_rules]

```

The result is shown below.

```

- OK..
1 subgoal:
> val it =

  ~even (SUC n) ==> even (SUC n + 1)
  -----
  ~even n ==> even (n + 1)
: proof

```

36

3. We simplify the goal as much as possible using REPEAT STRIP\_TAC.

```

REPEAT STRIP_TAC

```

The result is shown below.

```

- OK..
1 subgoal:
> val it =

  even (SUC n + 1)
  -----
  0. ~even n ==> even (n + 1)
  1. ~even (SUC n)
: proof

```

37

4. Looking at the subgoal and its assumptions, we see the following path to finish the proof. We note that the goal  $\text{even } (\text{SUC } n + 1)$  is the same as  $\text{even } (n + 2)$ . We have proved *even\_theorem*, which states  $\vdash \forall n. \text{even } n \iff \text{even } (n + 2)$ . Thus, we can prove the goal if we can prove  $\text{even } n$ . Looking at the assumptions, we can derive  $\text{even } n$  by what amounts to resolution (modus tollens) on the assumptions. This requires rewriting with  $\text{SUC } n = n + 1$ , which is the *ADD1* theorem,  $\vdash \forall m. \text{SUC } m = m + 1$ . *PROVE\_TAC*, augmented with the above theorems, finishes the proof.

```

PROVE_TAC[even_theorem, ADD1, (ARITH_CONV ``SUC n + 1 = n + 2``)]

```

The result is shown below.

```

- OK..
Meson search level: .....

Goal proved.
[.] |- even (SUC n + 1)

Goal proved.
[.] |- ~even (SUC n) ==> even (SUC n + 1)
> val it =
  Initial goal proved.
  |- ~even n ==> even (n + 1)
  : proof

```

38

5. Finally, we add the following code to *hlsm1Script.sml* and compile it with Holmake within a terminal.

```

val not_even_lemma =
  TAC_PROOF([[] , ``~even n ==> even (n+1) ``),
  Induct_on `n` THEN
  REWRITE_TAC[even_rules] THEN
  REPEAT STRIP_TAC THEN
  PROVE_TAC[even_theorem, ADD1, (ARITH_CONV ``SUC n + 1 = n + 2 ``)])

val _ = save_thm("not_even_lemma", not_even_lemma)

```

◇

### 12.2.10 not\_odd\_lemma

The theorem *not\_odd\_lemma* states that if  $n$  is not odd then  $n + 1$  must be odd. The proof is Exercise 12.3.5.

[not\_odd\_lemma]  
 $\vdash \neg \text{odd } n \Rightarrow \text{odd } (n + 1)$

### 12.2.11 even\_odd\_theorem

The theorem *even\_odd\_theorem* states that if  $n$  is even then  $n + 1$  is odd. This is a key theorem as it relates even and odd numbers  $n$  and  $n + 1$ .

[even\_odd\_theorem]  
 $\vdash \forall n. \text{even } n \Rightarrow \text{odd } (n + 1)$

The proof is straightforward recalling the following two theorems we have already proved.

[even\_not\_odd]  
 $\vdash \forall n. \text{even } n \Rightarrow \neg \text{odd } n$

[not\_odd\_lemma]  
 $\vdash \neg \text{odd } n \Rightarrow \text{odd } (n + 1)$

### Example 12.12

We prove *even\_odd\_theorem*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``!n.even n ==> odd (n + 1) ``);
```

The result is shown below.

```
> val it =
  Proof manager status: 2 proofs.
  2. Completed goalstack: |- ~even n ==> even (n + 1)
  1. Incomplete goalstack:
    Initial goal:

      !n. even n ==> odd (n + 1)

: proofs
```

39

2. We finish the proof with `PROVE_TAC` augmented with *even\_not\_odd*, and *not\_odd\_lemma*.

```
PROVE_TAC[even_not_odd,not_odd_lemma]
```

The result is shown below.

```
- OK..
Meson search level: ....
> val it =
  Initial goal proved.
  |- !n. even n ==> odd (n + 1)
  : proof
```

40

3. Finally, we add the code below to *hlsmlScript.sml* and compile it with `Holmake` within a terminal.

```
val even_odd_theorem =
  TAC_PROOF(([], ``!n.even n ==> odd (n + 1) ``),
    PROVE_TAC[even_not_odd,not_odd_lemma])

val _ = save_thm("even_odd_theorem",even_odd_theorem)
```

◇

### 12.2.12 odd\_even\_theorem

The theorem *odd\_even\_theorem* states that if  $n$  is odd then  $n + 1$  is even. The proof is Exercise 12.3.6.

[[odd\\_even\\_theorem](#)]  
 $\vdash \forall n. \text{odd } n \Rightarrow \text{even } (n + 1)$

### 12.2.13 evenEven\_lemma

The theorem *evenEven\_lemma* relates *even* to *Even*. It states that if *even*  $n$  is true, then *Even*  $n$  must be true, too.

[[evenEven\\_lemma](#)]  
 $\vdash \forall n. \text{even } n \Rightarrow \text{Even } n$

**Example 12.13**

We prove *evenEven\_lemma*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``!n. (even n) ==> (Even n) ``);
```

The result is shown below.

```
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !n. even n ==> Even n

: proofs
```

41

2. The form of the goal allows us to use rule induction on *even*.

```
Induct_on `even`
```

The result is shown below.

```
- OK..
1 subgoal:
> val it =

  Even 0 /\ !n. even n /\ Even n ==> Even (n + 2)

: proof
```

42

3. The goal is proved using PROVE\_TAC augmented by *Even\_rules*.

```
PROVE_TAC[Even_rules]
```

The result is shown below.

```
- OK..
Meson search level: .....

Goal proved.
|- Even 0 /\ !n. even n /\ Even n ==> Even (n + 2)
> val it =
  Initial goal proved.
  |- !n. even n ==> Even n
: proof
```

43

4. Finally, we add the following code to *hlsmlScript.sml* and compile it with Holmake in a terminal.

```
val evenEven_lemma =
  TAC_PROOF([], ``!n. (even n) ==> (Even n) ``),
  Induct_on `even` THEN
  PROVE_TAC[Even_rules])

val _ = save_thm("evenEven_lemma", evenEven_lemma)
```

◇

### 12.2.14 oddOdd\_lemma

The theorem *oddOdd\_lemma* states that if *odd n* is true then *Odd n* must be true, too. The proof is Exercise 12.3.7.

[oddOdd\_lemma]  
 $\vdash \forall n. \text{odd } n \Rightarrow \text{Odd } n$

### 12.2.15 Even\_even\_Odd\_odd\_lemma

The previous theorems show that *even n*  $\Rightarrow$  *Even n* and *odd n*  $\Rightarrow$  *Odd n*. To show that *even* is equivalent to *Even* and *odd* is equivalent to *Odd*, we need to show *Even n*  $\Rightarrow$  *even n* and *Odd n*  $\Rightarrow$  *odd n*. We proved these theorems separately.

Recall that *Even* and *Odd* are defined in terms of each other, as is their corresponding rule induction theorem *EvenOdd\_induction*.

[EvenOdd\_induction]  
 $\vdash \forall \text{Even}' \text{ Odd}'. \text{Even}' 0 \wedge (\forall n. \text{Odd}' n \Rightarrow \text{Even}' (n + 1)) \wedge (\forall n. \text{Even}' n \Rightarrow \text{Odd}' (n + 1)) \Rightarrow$   
 $(\forall a_0. \text{Even } a_0 \Rightarrow \text{Even}' a_0) \wedge \forall a_1. \text{Odd } a_1 \Rightarrow \text{Odd}' a_1$

The form of the above induction theorem means that to take advantage of rule induction over *Even* or *Odd*, our goal should be the conjunction  $(\text{Even } n \Rightarrow \text{even } n) \wedge (\text{Odd } n \Rightarrow \text{odd } n)$ .

The antecedent of the induction theorem corresponds to theorems already proved:

1. *even\_rules*:  $\vdash \text{even } 0 \wedge \forall n. \text{even } n \Rightarrow \text{even } (n + 2)$ ,
2. *odd\_even\_theorem*:  $\vdash \forall n. \text{odd } n \Rightarrow \text{even } (n + 1)$ , and
3. *even\_odd\_theorem*:  $\vdash \forall n. \text{even } n \Rightarrow \text{odd } (n + 1)$ .

### Example 12.14

We prove *Even\_even\_Odd\_odd\_lemma*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``(!n. (Even n) ==> (even n)) /\ (!n. (Odd n) ==> (odd n)) ``);
```

The result is shown below.

```
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      (!n. Even n ==> even n) /\ !n. Odd n ==> odd n

  : proofs
```

44

2. We invoke rule induction on *Even* by *Induct\_on* 'Even'.

```
Induct_on 'Even'
```

The result is shown below.

```

- OK..
1 subgoal:
> val it =

  even 0 /\ (!n. Odd n /\ odd n ==> even (n + 1)) /\
  !n. Even n /\ even n ==> odd (n + 1)

  : proof

```

45

3. We finish the prove with `PROVE_TAC` augmented with *even\_rules*, *even\_odd\_theorem*, and *odd\_even\_theorem*.

```
PROVE_TAC[even_rules,even_odd_theorem,odd_even_theorem]
```

The result is shown below.

```

- OK..
Meson search level: .....

Goal proved.
|- even 0 /\ (!n. Odd n /\ odd n ==> even (n + 1)) /\
  !n. Even n /\ even n ==> odd (n + 1)
> val it =
  Initial goal proved.
  |- (!n. Even n ==> even n) /\ !n. Odd n ==> odd n
  : proof

```

46

4. Finally, we add the following code to *hlsmlScript.sml* and compile it with `Holmake` in a terminal.

```

val Even_even_Odd_odd_lemma =
  TAC_PROOF([], ``(!n. (Even n) ==> (even n)) /\ (!n. (Odd n) ==> (odd n)) ``),
  Induct_on `Even` THEN
  PROVE_TAC[even_rules,even_odd_theorem,odd_even_theorem])

val _ = save_thm("Even_even_Odd_odd_lemma", Even_even_Odd_odd_lemma)

```

◇

### 12.2.16 even\_is\_Even

With the theorems *evenEven\_lemma* and *Even\_even\_Odd\_odd\_lemma* we prove that *even n* and *Even n* are logically equivalent.

#### Example 12.15

We prove *even\_is\_Even*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``!n. even n = Even n``);
```

The result is shown below.

```

> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

    !n. even n <=> Even n

  : proofs

```

47



2. The proof is done with `PROVE_TAC` augmented with *evenEven\_lemma* and *Even\_even\_Odd\_odd\_lemma*.

```
PROVE_TAC[evenEven_lemma, Even_even_Odd_odd_lemma]
```

The result is shown below.

```
- OK..
Meson search level: .....
> val it =
  Initial goal proved.
  |- !n. even n <=> Even n
  : proof
```

48

Finally, we add the following code to *hlsmlScript.sml* and compile it with `Holmake` in a terminal.

```
val even_is_Even =
  TAC_PROOF([], ``!n. even n = Even n``,
    PROVE_TAC[evenEven_lemma, Even_even_Odd_odd_lemma])

val _ = save_thm("even_is_Even", even_is_Even)
```

◇

### 12.2.17 odd is Odd

The theorem *odd\_is\_Odd* states that *odd n* is true if and only if *Odd n* is true, i.e., both are logically equivalent to each other.

[odd\_is\_Odd]  
 $\vdash \forall n. \text{odd } n \iff \text{Odd } n$

The proof is Exercise 12.3.8.

### 12.2.18 even is EVEN

This theorem shows the equivalence of *even n* to *EVEN n*, which is part of *arithmeticTheory* in HOL. Specifically, we prove the theorem

[even\_is\_EVEN]  
 $\vdash \forall n. \text{EVEN } n \iff \text{even } n$

Recall the definition of *EVEN* is as follows.

[EVEN]  
 $\vdash (\text{EVEN } 0 \iff \text{True}) \wedge \forall n. \text{EVEN } (\text{SUC } n) \iff \neg \text{EVEN } n$

Based on the above definition of *EVEN*, we prove that *even* has the exact same defining property as *EVEN*. Specifically, we prove:

[even\_not\_even\_thm]  
 $\vdash \neg \text{even } n \iff \text{even } (\text{SUC } n)$

We prove the above by proving each of the conditionals that constitute the biconditional.

```
[even_not_even_lemma1]
```

```
⊢ even (SUC n) ⇒ ¬even n
```

```
[even_not_even_lemma2]
```

```
⊢ ¬even n ⇒ even (SUC n)
```

### Example 12.16

We prove *even\_not\_even\_lemma1*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``even (SUC n) ==> ~even n``);
```

The result is shown below.

```
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      even (SUC n) ==> ~even n

: proofs
```

49

2. We use induction on the natural number  $n$ .

```
Induct_on `n`
```

The result is shown below.

```
- OK..
2 subgoals:
> val it =

  even (SUC (SUC n)) ==> ~even (SUC n)
  -----
  even (SUC n) ==> ~even n

  even (SUC 0) ==> ~even 0

2 subgoals
: proof
```

50

3. We see that the first subgoal is true because 1 is not even. We see that the second subgoal is true because  $even(n+2) = even\ n$ , as stated in *even\_theorem*, and the assumption  $even(SUC\ n) \Rightarrow \neg even\ n$ , whose contrapositive is  $even\ n \Rightarrow \neg even(SUC\ n)$ . We apply *PROVE\_TAC* augmented with *ADD\_CLAUSES*, *not\_even\_1*, *even\_theorem*,  $SUC(SUC\ n) = n + 2$ , and  $SUC\ n = n + 1$ .

```
PROVE_TAC[ADD_CLAUSES,not_even_1,even_theorem,
  (ARITH_CONV``SUC (SUC n) = n + 2``),
  (ARITH_CONV``SUC n = n + 1``)]
```

The result is shown below.

```

- OK..
Meson search level: .....

Goal proved.
|- even (SUC 0) ==> ~even 0

Remaining subgoals:
> val it =

    even (SUC (SUC n)) ==> ~even (SUC n)
    -----
    even (SUC n) ==> ~even n
    : proof

```

51

We apply PROVE\_TAC again augmented with the same theorems to prove the remaining subgoal.

```

PROVE_TAC[ADD_CLAUSES,not_even_1,even_theorem,
  (ARITH_CONV``SUC (SUC n) = n + 2``),
  (ARITH_CONV``SUC n = n + 1``)]

```

The result is shown below.

```

- OK..
Meson search level: .....

Goal proved.
[.] |- even (SUC (SUC n)) ==> ~even (SUC n)
> val it =
  Initial goal proved.
  |- even (SUC n) ==> ~even n
  : proof

```

52

4. Finally, we add the following code to *hlsm1Script.sml* and compile it with Holmake in a terminal.

```

val even_not_even_lemmal =
  TAC_PROOF([[]], ``even (SUC n) ==> ~even n``,
    Induct_on 'n' THEN
    PROVE_TAC[ADD_CLAUSES,not_even_1,even_theorem,
      (ARITH_CONV``SUC (SUC n) = n + 2``),
      (ARITH_CONV``SUC n = n + 1``)])

```

◇

### Example 12.17

We prove *even\_not\_even\_lemma2*.

1. We open a HOL session, load and open *hlsm1Theory*, *numLib*, and *arithmeticTheory*. We set the goal.

```

set_goal([], ``~even n ==> even (SUC n)``);

```

The result is shown below.

```

- OK..
2 subgoals:
> val it =

  ~even (SUC n) ==> even (SUC (SUC n))
  -----
  ~even n ==> even (SUC n)

  ~even 0 ==> even (SUC 0)

2 subgoals
: proof

```

53

2. We see that the first subgoal is true because 0 is even. We see that the second subgoal is true because  $\text{even}(n+2) = \text{even } n$ , as stated in *even\_theorem*, and the assumption  $\neg \text{even } n \Rightarrow \text{even } (\text{SUC } n)$ , whose contrapositive is  $\neg \text{even } (\text{SUC } n) \Rightarrow \text{even } n$ . We apply *PROVE\_TAC* augmented with *not\_even\_1*, *even\_theorem*, *SUC (SUC n) = n + 2*, and *SUC n = n + 1*.

```

PROVE_TAC[even_rules,even_theorem,
  (ARITH_CONV``SUC (SUC n) = n + 2``),
  (ARITH_CONV``SUC n = n + 1``)]

```

The result is shown below.

```

- OK..
Meson search level: ..

Goal proved.
|- ~even 0 ==> even (SUC 0)

Remaining subgoals:
> val it =

  ~even (SUC n) ==> even (SUC (SUC n))
  -----
  ~even n ==> even (SUC n)
: proof

```

54

We apply *PROVE\_TAC* again augmented by the same theorems.

```

PROVE_TAC[even_rules,even_theorem,
  (ARITH_CONV``SUC (SUC n) = n + 2``),
  (ARITH_CONV``SUC n = n + 1``)]

```

The result is shown below.

```

- OK..
Meson search level: .....

Goal proved.
[.] |- ~even (SUC n) ==> even (SUC (SUC n))
> val it =
  Initial goal proved.
  |- ~even n ==> even (SUC n)
: proof

```

55

3. Finally, we add the following code to *hlsmlScript.sml* and compile it with Holmake in a terminal.

```

val even_not_even_lemma2 =
TAC_PROOF([[]], ``~even n ==> even (SUC n) ``),
Induct_on `n` THEN
PROVE_TAC[even_rules, even_theorem,
  (ARITH_CONV``SUC (SUC n) = n + 2``),
  (ARITH_CONV``SUC n = n + 1``)]

```

◇

**Example 12.18**

We prove *even\_not\_even\_thm*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``~even n = even (SUC n) ``);
```

The result is shown below.

```

> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      ~even n <=> even (SUC n)

: proofs

```

56

2. We augment PROVE\_TAC with the theorems corresponding to both conditionals constituting the bi-conditional.

```
PROVE_TAC[even_not_even_lemma1, even_not_even_lemma2]
```

The result is shown below.

```

- OK..
Meson search level: .....
> val it =
  Initial goal proved.
  |- ~even n <=> even (SUC n)
  : proof

```

57

3. Finally, we add the following code to *hlsmlScript.sml* and compile it with `Holmake` in a terminal.

```

val even_not_even_thm =
TAC_PROOF([[]], ``~even n = even (SUC n) ``),
PROVE_TAC[even_not_even_lemma1, even_not_even_lemma2])

val _ = save_thm("even_not_even_thm", even_not_even_thm)

```

◇

**Example 12.19**

We prove *even\_is\_EVEN*.

1. We open a HOL session, load and open *hlsmlTheory*, *numLib*, and *arithmeticTheory*. We set the goal.

```
set_goal([], ``!n. EVEN n = even n``);
```

The result is shown below.

```
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      !n. EVEN n <=> even n

: proofs
```

58

2. We induct on natural number  $n$ .

```
Induct_on 'n'
```

The result is shown below.

```
- OK..
2 subgoals:
> val it =

  EVEN (SUC n) <=> even (SUC n)
  -----
  EVEN n <=> even n

  EVEN 0 <=> even 0

2 subgoals
: proof
```

59

3. The first subgoal is proved using the defining definitions *even\_rules* and *EVEN*. The second subgoal depends on *EVEN* and *even\_not\_even\_thm*. We prove both subgoals using *PROVE\_TAC* augmented with *even\_rules*, *EVEN*, and *even\_not\_even\_thm*.

```
PROVE_TAC[even_rules,EVEN,even_not_even_thm]
```

The result is shown below.

```
- OK..
Meson search level: ....

Goal proved.
|- EVEN 0 <=> even 0

Remaining subgoals:
> val it =

  EVEN (SUC n) <=> even (SUC n)
  -----
  EVEN n <=> even n

: proof
```

60

We apply *PROVE\_TAC* again augmented by the same theorems.

```
PROVE_TAC[even_rules,EVEN,even_not_even_thm]
```

The result is shown below.

```
- OK..
Meson search level: .....

Goal proved.
[.] |- EVEN (SUC n) <=> even (SUC n)
> val it =
  Initial goal proved.
  |- !n. EVEN n <=> even n
  : proof
```

61

4. Finally, we add the following code to *hlsmlScript.sml* and compile it with `Holmake` in a terminal.

```
val even_is_EVEN =
  TAC_PROOF([[] , ``!n.EVEN n = even n``),
  Induct_on `n` THEN
  PROVE_TAC[even_rules,EVEN,even_not_even_thm])

val _ = save_thm("even_is_EVEN",even_is_EVEN)
```

◇

### 12.2.19 Even\_is\_EVEN

The theorem *Even\_is\_EVEN* states that *Even n* is logically equivalent to *EVEN n*.

```
[Even_is_EVEN]
⊢ ∀n. EVEN n ⇔ Even n
```

The proof is straightforward given we have proved that *Even* is logically equivalent to *even*. The proof is Exercise 12.3.9.

### 12.2.20 odd\_is\_ODD

The theorem *odd\_is\_ODD* states that *odd n* is logically equivalent to *ODD n*.

```
[odd_is_ODD]
⊢ ∀n. ODD n ⇔ odd n
```

The proof is Exercise 12.3.10.

### 12.2.21 Odd\_is\_ODD

The theorem *Odd\_is\_ODD* states that *Odd n* is logically equivalent to *ODD n*.

```
[Odd_is_ODD]
⊢ ∀n. ODD n ⇔ Odd n
```

The proof is Exercise 12.3.11.

## 12.3 Exercises

### Exercise 12.3.1

A. Prove that `Odd 1` is true.

$\vdash \text{Odd } 1$

B. Use the above theorem and prove the following theorem. Save it under the name `Odd_rules`.

`[Odd_rules]`

$\vdash \text{Odd } 1 \wedge \forall n. \text{Odd } n \Rightarrow \text{Odd } (n + 2)$

### Exercise 12.3.2

Prove that 1 is not odd, i.e.,

`[not_even_1]`

$\vdash \neg \text{even } 1$

### Exercise 12.3.3

Prove that if  $n$  is odd then  $n$  is not even. Name this theorem `odd_not_even`.

`[odd_not_even]`

$\vdash \forall n. \text{odd } n \Rightarrow \neg \text{even } n$

### Exercise 12.3.4

A. Prove that 2 is even, i.e.,

`[even2]`

$\vdash \text{even } 2$

B. Prove that 2 is not odd, i.e.,

`[not_odd2]`

$\vdash \neg \text{odd } 2$

C. Prove that if  $n + 2$  is odd, then  $n$  must be odd, i.e.,

`[odd_lemma1]`

$\vdash \forall n. \text{odd } (n + 2) \Rightarrow \text{odd } n$

D. Prove that  $n$  is odd if and only if  $n + 2$  is odd. Save the theorem under the name `odd_theorem`.

`[odd_theorem]`

$\vdash \forall n. \text{odd } n \iff \text{odd } (n + 2)$

### Exercise 12.3.5

Prove that if  $n$  is not odd then  $n + 1$  is odd. Name the theorem `not_odd_lemma`.

`[not_odd_lemma]`

$\vdash \neg \text{odd } n \Rightarrow \text{odd } (n + 1)$



**Exercise 12.3.6** Prove if  $n$  is odd then  $n + 1$  is even. Save it under the name `odd_even_theorem`.

`[odd_even_theorem]`  
 $\vdash \forall n. \text{odd } n \Rightarrow \text{even } (n + 1)$

**Exercise 12.3.7**

Prove that if `odd n` is true then so is `Odd n`. Save the theorem under the name `oddOdd_lemma`.

`[oddOdd_lemma]`  
 $\vdash \forall n. \text{odd } n \Rightarrow \text{Odd } n$

**Exercise 12.3.8**

Prove that `odd n = Odd n`. Save the theorem under the name `odd_is_Odd`.

`[odd_is_Odd]`  
 $\vdash \forall n. \text{odd } n \iff \text{Odd } n$

**Exercise 12.3.9**

Prove that `Even n` is logically equivalent to `EVEN n`.

`[Even_is_EVEN]`  
 $\vdash \forall n. \text{EVEN } n \iff \text{Even } n$

**Exercise 12.3.10**

Prove that `odd n` is logically equivalent to `ODD n`.

`[odd_is_ODD]`  
 $\vdash \forall n. \text{ODD } n \iff \text{odd } n$

Similar to the proof of `even.is.EVEN`, you may want to prove the following lemmas.

`[odd_not_odd_lemma1]`  
 $\vdash \text{odd } (\text{SUC } n) \Rightarrow \neg \text{odd } n$

`[odd_not_odd_lemma2]`  
 $\vdash \neg \text{odd } n \Rightarrow \text{odd } (\text{SUC } n)$

`[odd_not_odd_thm]`  
 $\vdash \neg \text{odd } n \iff \text{odd } (\text{SUC } n)$

**Exercise 12.3.11**

Prove that `Odd n` is logically equivalent to `ODD n`.

`[Odd_is_ODD]`  
 $\vdash \forall n. \text{ODD } n \iff \text{Odd } n$

BLANK PAGE

## **Part IV**

# **Lab Exercises: Access-Control Logic in HOL**



# An Access-Control Logic in HOL

This section describes an access-control logic that is a calculus for reasoning about authentication and authorization. Our description is brief for space considerations. A full account appears in [Chin and Older, 2010]. We present the syntax, semantics, and inference rules in the following sections. Of course, the syntax, semantics, and inference rules are fully implemented and verified in HOL.

## 13.1 Syntax

The syntax of the logic has two major components.

1. The syntax of *principals*, where principals are informally thought of as the actors making statements, e.g., people, cryptographic keys, userids and passwords associated with accounts, etc.
2. The syntax of logical formulas.

The syntax of principal expressions *Princ* is defined as follows.

$$\mathbf{Princ} ::= \mathbf{PName} / \mathbf{Princ} \ \& \ \mathbf{Princ} / \mathbf{Princ} \ | \ \mathbf{Princ}$$

“&” is pronounced “with”; “|” is pronounced “quoting”. The type of principal expressions is composed of principal names, e.g., Alice, cryptographic keys, and userid with passwords. Compound expressions are created with & and |.

Examples of principal expressions include

$$Alice \quad K_{Alice} \quad Alice \ \& \ Bob \quad Alice \ | \ Bob$$

Informally, *Alice* is Alice, *K<sub>Alice</sub>* is Alice’s cryptographic key, *Alice & Bob* is Alice and Bob together, *Alice | Bob* is Alice quoting Bob (relaying his statements).

The syntax of logical formulas *Form* consists of propositional variables, expressions using the usual propositional operators corresponding to *modal* versions of negation, conjunction, disjunction, implication, and equivalence, coupled with operators  $\Rightarrow$  (pronounced “speaks for”), *says*, *controls*, and *reps*.

In this presentation of the C2 calculus, we use the same symbols for negation, conjunction, disjunction, implication, and equivalence in propositional logic. In the HOL implementation of the access-control logic, negation, conjunction, disjunction, implication, and equivalence in the access-control logic are represented using different symbols to clearly distinguish between access-control logic formulas and propositional logic formulas.

$$\begin{aligned} \mathbf{Form} ::= & \mathbf{PropVar} / \neg \mathbf{Form} / \\ & (\mathbf{Form} \vee \mathbf{Form}) / (\mathbf{Form} \wedge \mathbf{Form}) / \\ & (\mathbf{Form} \supset \mathbf{Form}) / (\mathbf{Form} \equiv \mathbf{Form}) / \\ & (\mathbf{Princ} \Rightarrow \mathbf{Form}) / (\mathbf{Princ} \text{ says } \mathbf{Form}) / \\ & (\mathbf{Princ} \text{ controls } \mathbf{Form}) / \mathbf{Princ} \text{ reps } \mathbf{Princ} \text{ on } \mathbf{Form} \end{aligned}$$

**Figure 13.1** CONOPS Statements and Their Representation in the C2 Calculus

C2 Statement	Formula
If $\phi_1$ is true then $\phi_2$ is true (typical of policy statements)	$\phi_1 \supset \phi_2$
Key associated with Alice	$K_a \Rightarrow \text{Alice}$
Bob has jurisdiction (controls or is believed) over statement $\phi$	<i>Bob controls</i> $\phi$
Alice and Bob together say $\phi$	<i>(Alice &amp; Bob) says</i> $\phi$
Alice quotes Bob as saying $\phi$	<i>(Alice   Bob) says</i> $\phi$
Bob is Alice's delegate on statement $\phi$	<i>Bob reps Alice</i> on $\phi$
Carol is authorized in Role on statement $\phi$	<i>Carol reps Role</i> on $\phi$
Carol acting in Role makes statement $\phi$	<i>(Carol   Role) says</i> $\phi$

**Figure 13.2** Kripke Semantics of Access-Control Logic Formulas

$$\begin{aligned}
\mathcal{E}_{\mathcal{M}}[p] &= I(p) \\
\mathcal{E}_{\mathcal{M}}[\neg\phi] &= W - \mathcal{E}_{\mathcal{M}}[\phi] \\
\mathcal{E}_{\mathcal{M}}[\phi_1 \wedge \phi_2] &= \mathcal{E}_{\mathcal{M}}[\phi_1] \cap \mathcal{E}_{\mathcal{M}}[\phi_2] \\
\mathcal{E}_{\mathcal{M}}[\phi_1 \vee \phi_2] &= \mathcal{E}_{\mathcal{M}}[\phi_1] \cup \mathcal{E}_{\mathcal{M}}[\phi_2] \\
\mathcal{E}_{\mathcal{M}}[\phi_1 \supset \phi_2] &= (W - \mathcal{E}_{\mathcal{M}}[\phi_1]) \cup \mathcal{E}_{\mathcal{M}}[\phi_2] \\
\mathcal{E}_{\mathcal{M}}[\phi_1 \equiv \phi_2] &= \mathcal{E}_{\mathcal{M}}[\phi_1 \supset \phi_2] \cap \mathcal{E}_{\mathcal{M}}[\phi_2 \supset \phi_1] \\
\mathcal{E}_{\mathcal{M}}[P \Rightarrow Q] &= \begin{cases} W, & \text{if } \hat{J}(Q) \subseteq \hat{J}(P) \\ \emptyset, & \text{otherwise} \end{cases} \\
\mathcal{E}_{\mathcal{M}}[P \text{ says } \phi] &= \{w \mid \hat{J}(P)(w) \subseteq \mathcal{E}_{\mathcal{M}}[\phi]\} \\
\mathcal{E}_{\mathcal{M}}[P \text{ controls } \phi] &= \mathcal{E}_{\mathcal{M}}[(P \text{ says } \phi) \supset \phi] \\
\mathcal{E}_{\mathcal{M}}[P \text{ reps } Q \text{ on } \phi] &= \mathcal{E}_{\mathcal{M}}[(P \mid Q \text{ says } \phi) \supset Q \text{ says } \phi]
\end{aligned}$$

Figure 13.1 is a table of typical C2 statements and their representation as formulas in the C2 calculus.

## 13.2 Semantics

The semantics of the access-control logic uses *Kripke structures*. 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*.
- $I : \mathbf{PropVar} \rightarrow \mathcal{P}(W)$  is an *interpretation* function that maps each propositional variable  $p$  to a set of worlds.
- $J : \mathbf{PName} \rightarrow \mathcal{P}(W \times W)$  is a function that maps each principal name  $A$  into a relation on worlds (i.e., a subset of  $W \times W$ ).

The semantics of principal expressions *Princ* involves  $J$  and its extension  $\hat{J}$ . We define the extended function  $\hat{J} : \mathbf{Princ} \rightarrow \mathcal{P}(W \times W)$  inductively on the structure of principal expressions, where  $A \in \mathbf{PName}$ .

$$\begin{aligned}
\hat{J}(A) &= J(A) \\
\hat{J}(P \& Q) &= \hat{J}(P) \cup \hat{J}(Q) \\
\hat{J}(P \mid Q) &= \hat{J}(P) \circ \hat{J}(Q).
\end{aligned}$$

Note:  $R_1 \circ R_2 = \{(x, z) \mid \exists y. (x, y) \in R_1 \text{ and } (y, z) \in R_2\}$ .

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

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

where  $\mathcal{E}_{\mathcal{M}}[\![ \varphi ]\!]$  is the set of worlds in which  $\varphi$  is considered true.

$\mathcal{E}_{\mathcal{M}}[\![ \varphi ]\!]$  is defined inductively on the structure of  $\varphi$ , as shown in Figure 13.2. Note, in the definition of  $\mathcal{E}_{\mathcal{M}}[\![ P \text{ says } \varphi ]\!]$ , that  $\hat{J}(P)(w)$  is simply the image of world  $w$  under the relation  $\hat{J}(P)$ .

### 13.3 Inference Rules

**Figure 13.3** Inference rules for the access-control logic

---

$P \text{ controls } \varphi \stackrel{\text{def}}{=} (P \text{ says } \varphi) \supset \varphi$		$P \text{ reps } Q \text{ on } \varphi \stackrel{\text{def}}{=} P \mid Q \text{ says } \varphi \supset Q \text{ says } \varphi$	
<i>Modus Ponens</i>	$\frac{\varphi \quad \varphi \supset \varphi'}{\varphi'}$	<i>Says</i>	$\frac{\varphi}{P \text{ says } \varphi}$
		<i>Controls</i>	$\frac{P \text{ controls } \varphi \quad P \text{ says } \varphi}{\varphi}$
<i>Derived Speaks For</i>	$\frac{P \Rightarrow Q \quad P \text{ says } \varphi}{Q \text{ says } \varphi}$	<i>Reps</i>	$\frac{Q \text{ controls } \varphi \quad P \text{ reps } Q \text{ on } \varphi \quad P \mid Q \text{ says } \varphi}{\varphi}$
<i>&amp; Says (1)</i>	$\frac{P \ \& \ Q \text{ says } \varphi}{P \text{ says } \varphi \ \& \ Q \text{ says } \varphi}$	<i>&amp; Says (2)</i>	$\frac{P \text{ says } \varphi \ \& \ Q \text{ says } \varphi}{P \ \& \ Q \text{ says } \varphi}$
<i>Quoting (1)</i>	$\frac{P \mid Q \text{ says } \varphi}{P \text{ says } Q \text{ says } \varphi}$	<i>Quoting (2)</i>	$\frac{P \text{ says } Q \text{ says } \varphi}{P \mid Q \text{ says } \varphi}$
<i>Idempotency of <math>\Rightarrow</math></i>	$\frac{}{P \Rightarrow P}$	<i>Monotonicity of <math>\mid</math></i>	$\frac{P' \Rightarrow P \quad Q' \Rightarrow Q}{P' \mid Q' \Rightarrow P \mid Q}$

---

Our use of the access-control logic as a C2 calculus rarely, if ever, uses Kripke structures explicitly. Instead, we rely upon inference rules to derive expressions soundly.

An inference rule in the C2 calculus has the form

$$\frac{H_1 \quad \dots \quad H_k}{C},$$

where  $H_1 \dots H_k$  is a (possibly empty) set of *hypotheses* expressed as access-control logic formulas, and  $C$  is the *conclusion*, also expressed as an access-control logic formula. Whenever all of the hypotheses in an inference rule are present in a proof, then the rule states it is permissible to include the conclusion in the proof, too.

The meaning of *sound* depends on the the definition of *satisfies* in the access-control logic. A Kripke structure  $\mathcal{M}$  **satisfies** a formula  $\varphi$  when  $\mathcal{E}_{\mathcal{M}}[\![ \varphi ]\!] = W$ , i.e.,  $\varphi$  is true in all worlds  $W$  of  $\mathcal{M}$ . We denote  $\mathcal{M}$  satisfies  $\varphi$  by  $\mathcal{M} \models \varphi$ .

A C2 calculus inference rule is **sound** if, for all Kripke structures  $\mathcal{M}$ , whenever  $\mathcal{M}$  satisfies all the hypotheses  $H_1 \dots H_k$ , then  $\mathcal{M}$  also satisfies  $C$ , i.e., if for all  $\mathcal{M}$ :  $\mathcal{M} \models H_i$  for  $1 \leq i \leq k$ , then it must be the case that  $\mathcal{M} \models C$ .

All the inference rules presented here and in [Chin and Older, 2010] are proved to be logically sound. Figure 13.3 are the core inference rules of the access-control logic.

### 13.4 Describing Access-Control Concepts in the C2 Calculus

To illustrate how the C2-calculus is used to reason about authentication and authorization, we consider the following use case.

#### Example 13.1

Bob guards access to sensitive files. He receives requests electronically and says yes or no to each request. Specifically, the requests he receives are digitally signed by a cryptographic key. Keys are associated with people, e.g., Alice. If the person, say Alice, who owns the key has permission to access the file, then Bob says yes.

Suppose Bob receives an access request signed by Alice's key  $K_A$ , and that Alice is permitted to access the files. We represent the request, the link between Alice and her key  $K_A$ , and her permission to access the files by the following statements in the access-control logic.

1. Digitally signed request received by Bob:  $K_A$  says  $\langle \text{access files} \rangle$ .
2.  $K_A$  is Alice's key:  $K_A \Rightarrow \text{Alice}$ .
3. Alice has permission to access the files:  $\text{Alice}$  controls  $\langle \text{access files} \rangle$

Using the inference rules of the C2 calculus, Bob justifies his decision to grant Alice's request by the following proof, where lines 1–3 are the assumptions, and everything that follows is derived using the inference rules of the C2 calculus.

- |  |                                    |
|--|------------------------------------|
| 1. $K_A$ says $\langle \text{access files} \rangle$              | Digitally signed request           |
| 2. $K_A \Rightarrow \text{Alice}$                                | Key associated with Alice          |
| 3. $\text{Alice}$ controls $\langle \text{access files} \rangle$ | Alice's capability to access files |
| 4. $\text{Alice}$ says $\langle \text{access files} \rangle$     | 2, 1 Derived Speaks For            |
| 5. $\langle \text{access files} \rangle$                         | 3, 4 Controls                      |

Line 4 amounts to authenticating that Alice is the originator of the access request within the context established by lines 1 through 3. Line 3 establishes Alice's authority to access the files. Line 5 is Bob's deduction that granting Alice access is justified.

As a result of the proof, Bob has a derived inference rule, which he knows is sound because he derived it using the inference rules in Figure 13.3. The derived inference rule is

$$\frac{K_A \text{ says } \langle \text{access files} \rangle \quad K_A \Rightarrow \text{Alice} \quad \text{Alice controls } \langle \text{access files} \rangle}{\langle \text{access files} \rangle}$$

The inference rule amounts to a checklist. If he (1) gets a message cryptographically signed with  $K_A$ , (2)  $K_A$  is Alice's key, and (3) Alice has permission to access the files, then granting access to Alice is justified.

Looking back at Figure 14.1, the inference rule is a logically sound description of what Bob does in the top-level CONOPS. The inference rule makes explicit the policies and trust assumptions and how they combine to justify Bob's actions.  $\diamond$

Delegation is widely used. Our definition of delegation is given by the definition of *reps* and the *Reps* inference rule.

$$\text{Reps} \quad \frac{P \text{ reps } Q \text{ on } \varphi \stackrel{\text{def}}{=} P \mid Q \text{ says } \varphi \supset Q \text{ says } \varphi \quad Q \text{ controls } \varphi \quad P \text{ reps } Q \text{ on } \varphi \quad P \mid Q \text{ says } \varphi}{\varphi}$$



The consequence of the definition of *reps* in the first formula shows is this: if you believe *Alice* reps *Bob* on  $\phi$  is true, then if Alice says Bob says  $\phi$  you will conclude that Bob says  $\phi$ . In other words, Alice is trusted when she says Bob says  $\phi$ .

In a command and control application, if you believe (1) Bob is authorized on command  $\phi$ , (2) Alice is Bob's delegate or representative on a command  $\phi$ , and (3) Alice says Bob says command  $\phi$ , then you are justified to conclude the command  $\phi$  is legitimate. This is the *Reps* inference rule.

*Reps* is particularly useful for delegating limited authority to delegates. Unlike  $\Rightarrow$ , where all statements of one principal are attributable to another, *Reps* specifies which statements made by a delegate are attributable to another.

*Reps* is used when people are acting in defined roles, for example the roles of *Commander* and *Operator*. The following example show the use of *reps* in the context of roles.

### Example 13.2

Suppose we have two roles, two people, and two commands. The roles are *Commander* and *Operator*; the people are Alice and Bob; the two commands are *go* and *launch*. A *Commander* has the authority to issue a *go* command. An *Operator* has the authority to issue a *launch* command whenever a *go* command is received from a *Commander*. *Commanders* are not authorized to *launch*. *Operators* are not authorized to *launch* unless they receive a *go* command.

In this scenario, *Alice* is the *Commander* and *Bob* is an *Operator*. Notice that this scenario is captured by Figure 14.1.

We represent the notion that *Alice* and *Bob* are acting in their assigned roles of *Commander* and *Operator* using quotation and delegation. With Figure 14.1 in mind, we do the following analysis from Bob's perspective.

1. Message Bob receives signed by Alice's key:

$$K_A \mid \text{Commander says } \langle go \rangle$$

2. Bob's belief that  $K_A$  is Alice's key:

$$K_A \Rightarrow \text{Alice}$$

3. Bob's recognition that Alice is acting as *Commander* when issuing a *go* command:

$$\text{Alice reps Commander on } \langle go \rangle.$$

4. Bob's belief that *Commanders* have authority to issue *go* commands:

$$\text{Commander controls } \langle go \rangle.$$

5. The policy guiding Bob's actions, when he authenticates and authorizes a *go* command, then he is to issue a *launch* command:

$$\langle go \rangle \supset \langle launch \rangle$$

The input in line 1 with the other 4 assumptions as security context for Bob's decision is sufficient for Bob to issue the command  $K_B \mid \text{Operator says } \langle launch \rangle$ . The proof is as follows using the inference rules in Figure 13.3.

1. $K_A \mid \text{Commander says } \langle go \rangle$	Input signed by $K_A$
2. $K_A \Rightarrow \text{Alice}$	Trust assumption— $K_A$ is Alice's key
3. $\text{Alice reps Commander on } \langle go \rangle$	Trust assumption—Alice is acting as a Commander when issuing a <i>go</i> command
4. $\text{Commander controls } \langle go \rangle$	Trust assumption—Commanders have authority to issue a <i>go</i> command
5. $\langle go \rangle \supset \langle launch \rangle$	Policy assumption—if <i>go</i> is true then so is <i>launch</i>
6. $\text{Commander} \Rightarrow \text{Commander}$	Idempotency of $\Rightarrow$
7. $K_A \mid \text{Commander} \Rightarrow \text{Alice} \mid \text{Commander}$	2, 6 Monotonicity of $\mid$
8. $\text{Alice} \mid \text{Commander says } \langle go \rangle$	7, 1 Derived Speaks For
9. $\langle go \rangle$	4, 3, 8 Reps
10. $\langle launch \rangle$	9, 5 Modus Ponens
11. $K_B \mid \text{Operator says } \langle launch \rangle$	10 Says

The above proof justifies a derived inference rule showing the soundness of Bob's actions:

$$\frac{
 \begin{array}{l}
 K_A \mid \text{Commander says } \langle go \rangle \\
 K_A \Rightarrow \text{Alice} \quad \text{Alice reps Commander on } \langle go \rangle \\
 \text{Commander controls } \langle go \rangle \quad \langle go \rangle \supset \langle launch \rangle
 \end{array}
 }{
 K_B \mid \text{Operator says } \langle launch \rangle
 }$$

The derived inference rule is a logical checklist. If (1) *Bob* receives a cryptographically signed message using key  $K_A$  issuing a *go* order while quoting a *Commander* role, (2)  $K_A$  is Alice's key, (3) *Alice* is authorized to issue a *go* command as a *Commander*, (4) *Commanders* have the authority to issue a *go* command, and (5) the policy is when *go* is true the *launch* is true, then issuing  $K_B \mid \text{Operator says } \langle launch \rangle$  is justified, where  $K_B$  is Bob's key.  $\diamond$

We now turn our attention to automated support for reasoning using the HOL theorem prover for the access-control logic in the next section.

## 13.5 The Access-Control Logic in HOL

The access-control logic described in Sections 13.1, 13.2, and 13.3 is implemented in HOL by (1) defining its syntax as an algebraic type *Form*, (2) inductively defining the semantic function  $\mathcal{E}_{\mathcal{M}}[\![ - ]\!]$  in HOL over the type *Form* of access-control logic formulas, and (3) proving theorems in HOL corresponding to inference rules of the C2 calculus.

The benefits of implementing the access-control logic in HOL include:

1. complete disclosure of all access-control logic and C2 calculus syntax and semantics,
2. formal machine-checked proofs of all properties of the access-control logic,
3. quantification over access-control logic formulas,
4. ability to combine the access-control logic with other logical descriptions, and
5. rapid and easy reproduction of all results by third parties.

Sections 13.6, 13.7, and 13.8 describe the syntax, semantics, and theorems corresponding to the inference rules of the access-control logic and C2 calculus, respectively.

**Figure 13.4** Access-Control Logic Syntax in HOL

---

```

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

```

---

## 13.6 Syntax of the Access-Control Logic in HOL

The access-control logic is implemented as a conservative extension to the HOL system. What this means is that the HOL logic is extended by defining the *Form* algebraic type corresponding to access-control logic formulas, the algebraic type *Princ* corresponding to principal expressions, and the algebraic type *Kripke* corresponding to Kripke structures. The semantics of *Form* and *Princ* are defined using *Kripke* and existing HOL operators. The properties of the access-control logic are proved as theorems in HOL.

Figure 13.4 shows the HOL type *Form* corresponding to access-control logic formulas in HOL. Notice that the HOL implementation uses *notf*, *andf*, *orf*, *impf* and *eqf* to represent negation, conjunction, disjunction, implication, and equivalence in the access-control logic. Their semantics is defined in terms of sets of worlds from the universe of worlds that is part of a Kripke structure  $\mathcal{M}$ . This is different than the semantics of the corresponding operators in propositional logic. The propositional logic operators are defined in terms of truth values instead of sets of worlds.

The type definition in Figure 13.4 is polymorphic, i.e., allows for type substitution into type variables. Recall that type variables in HOL start with the back-quote symbol `'`. For example, atomic propositions in the access-control logic in HOL start with the type constructor `prop` and are applied to any type, as represented by `'aavar`. For example, `prop command` takes elements of the type `command` and maps them to propositions in the access-control logic in HOL.

Figure 13.5 shows the syntax of principal expressions, integrity and security labels, and Kripke structures in HOL. The HOL implementation parameterizes security labels, integrity labels, and their partial orders. As our thermostat example does not rely upon security or integrity labels, we will not discuss their use further.

**Figure 13.5** Syntax of Principal Expressions, Integrity and Security Labels, and Kripke Structures in HOL

---

```

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

IntLevel = iLab 'il | il 'apn ;

SecLevel = sLab 'sl | sl 'apn

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

```

---

**Figure 13.6** C2 Formulas and Their Representation in HOL

---

C2 Formula	HOL Syntax
$\langle \text{jump} \rangle$	prop jump
$\neg \langle \text{jump} \rangle$	notf (prop jump)
$\langle \text{run} \rangle \wedge \langle \text{jump} \rangle$	prop run andf prop jump
$\langle \text{run} \rangle \vee \langle \text{stop} \rangle$	prop run orf prop stop
$\langle \text{run} \rangle \supset \langle \text{jump} \rangle$	prop run impf prop jump
$\langle \text{walk} \rangle \equiv \langle \text{stop} \rangle$	prop walk eqf prop stop
<i>Alice</i> says $\langle \text{jump} \rangle$	Name Alice says prop jump
<i>Alice &amp; Bob</i> says $\langle \text{stop} \rangle$	Name Alice meet Name Bob says prop stop
<i>Bob   Carol</i> says $\langle \text{run} \rangle$	Name Bob quoting Name Carol says prop run
<i>Bob</i> controls $\langle \text{walk} \rangle$	Name Bob controls prop walk
<i>Bob</i> reps <i>Alice</i> on $\langle \text{jump} \rangle$	reps (Name Bob) (Name Alice) (prop jump)
<i>Carol</i> $\Rightarrow$ <i>Bob</i>	Name Carol speaks_for Name Bob

---

Examples using security and integrity labels are in [Chin and Older, 2010].

The type constructor *Name* is polymorphic as seen in the type definition of *Princ*, where it is applied to the type variable *'apn*. The infix type constructor *meet* corresponds to  $\&$ . The infix type constructor *quoting* corresponds to  $|$ .

Figure 13.6 is a table showing how formulas in the C2 calculus are written in HOL implementation of the access-control logic. The proposition  $\langle \text{jump} \rangle$  is written as *prop jump* in HOL. Negation of a C2 formula, such as  $\neg \langle \text{jump} \rangle$  is written as *notf (prop jump)* in HOL. *Alice* says  $\langle \text{jump} \rangle$  is written as *Name Alice says prop jump*, etc.

## 13.7 Semantics of the Access-Control Logic in HOL

With the introduction of logical expressions, principal expressions, and Kripke structures as datatypes into HOL, we can define the HOL function *Efn* corresponding to the function  $\mathcal{E}_{\mathcal{M}}[\![\!-\!]\!]$  in Figure 13.2, which defines the Kripke semantics of the access-control logic. The definition of *Efn* is below. The definitions of  $\mathcal{E}_{\mathcal{M}}[\![\!-\!]\!]$  and *Efn* closely correspond to one another syntactically.

Of course, the question is how do we know that the implementation in HOL corresponds to the logic

described in Figure 13.2 and as described in [Chin and Older, 2010]? The answer is if we can prove theorems in HOL about the HOL implementation that correspond to the inference rules in [Chin and Older, 2010], then we are satisfied.

The semantics or values of well-formed access-control logic formulas in HOL, is defined by  $Efn$ . The values of well-formed access-control logic formulas are sets of worlds that are members of the universe of worlds for a given Kripke structure  $M$ . The HOL definition of  $Efn$  appears below.

[Efn\_def]

$$\begin{aligned}
&\vdash (\forall Oi Os M. Efn Oi Os M TT = \mathcal{U}(:'v)) \wedge \\
&(\forall Oi Os M. Efn Oi Os M FF = \{\}) \wedge \\
&(\forall Oi Os M p. Efn Oi Os M (\text{prop } p) = \text{intpKS } M p) \wedge \\
&(\forall Oi Os M f. \\
&\quad Efn Oi Os M (\text{notf } f) = \mathcal{U}(:'v) \text{ DIFF } Efn Oi Os M f) \wedge \\
&(\forall Oi Os M f_1 f_2. \\
&\quad Efn Oi Os M (f_1 \text{ andf } f_2) = \\
&\quad Efn Oi Os M f_1 \cap Efn Oi Os M f_2) \wedge \\
&(\forall Oi Os M f_1 f_2. \\
&\quad Efn Oi Os M (f_1 \text{ orf } f_2) = \\
&\quad Efn Oi Os M f_1 \cup Efn Oi Os M f_2) \wedge \\
&(\forall Oi Os M f_1 f_2. \\
&\quad Efn Oi Os M (f_1 \text{ impf } f_2) = \\
&\quad \mathcal{U}(:'v) \text{ DIFF } Efn Oi Os M f_1 \cup Efn Oi Os M f_2) \wedge \\
&(\forall Oi Os M f_1 f_2. \\
&\quad Efn Oi Os M (f_1 \text{ eqf } f_2) = \\
&\quad (\mathcal{U}(:'v) \text{ DIFF } Efn Oi Os M f_1 \cup Efn Oi Os M f_2) \cap \\
&\quad (\mathcal{U}(:'v) \text{ DIFF } Efn Oi Os M f_2 \cup Efn Oi Os M f_1)) \wedge \\
&(\forall Oi Os M P f. \\
&\quad Efn Oi Os M (P \text{ says } f) = \\
&\quad \{w \mid \text{Jext } (jKS M) P w \subseteq Efn Oi Os M f\}) \wedge \\
&(\forall Oi Os M P Q. \\
&\quad Efn Oi Os M (P \text{ speaks\_for } Q) = \\
&\quad \text{if } \text{Jext } (jKS M) Q \text{ RSUBSET } \text{Jext } (jKS M) P \text{ then } \mathcal{U}(:'v) \\
&\quad \text{else } \{\}) \wedge \\
&(\forall Oi Os M P f. \\
&\quad Efn Oi Os M (P \text{ controls } f) = \\
&\quad \mathcal{U}(:'v) \text{ DIFF } \{w \mid \text{Jext } (jKS M) P w \subseteq Efn Oi Os M f\} \cup \\
&\quad Efn Oi Os M f) \wedge \\
&(\forall Oi Os M P Q f. \\
&\quad Efn Oi Os M (\text{reps } P Q f) = \\
&\quad \mathcal{U}(:'v) \text{ DIFF } \\
&\quad \{w \mid \text{Jext } (jKS M) (P \text{ quoting } Q) w \subseteq Efn Oi Os M f\} \cup \\
&\quad \{w \mid \text{Jext } (jKS M) Q w \subseteq Efn Oi Os M f\}) \wedge \\
&(\forall Oi Os M intl_1 intl_2. \\
&\quad Efn Oi Os M (intl_1 \text{ domi } intl_2) = \\
&\quad \text{if } \text{repPO } Oi (\text{Lifn } M intl_2) (\text{Lifn } M intl_1) \text{ then } \mathcal{U}(:'v) \\
&\quad \text{else } \{\}) \wedge \\
&(\forall Oi Os M intl_2 intl_1. \\
&\quad Efn Oi Os M (intl_2 \text{ eqi } intl_1) = \\
&\quad (\text{if } \text{repPO } Oi (\text{Lifn } M intl_2) (\text{Lifn } M intl_1) \text{ then } \mathcal{U}(:'v) \\
&\quad \text{else } \{\}) \cap \\
&\quad \text{if } \text{repPO } Oi (\text{Lifn } M intl_1) (\text{Lifn } M intl_2) \text{ then } \mathcal{U}(:'v) \\
&\quad \text{else } \{\}) \wedge \\
&(\forall Oi Os M secl_1 secl_2.
\end{aligned}$$

```

Efn Oi Os M (secl1 doms secl2) =
  if repPO Os (Lsfm M secl2) (Lsfm M secl1) then U(:'v)
  else {} ) ∧
(∀ Oi Os M secl2 secl1.
  Efn Oi Os M (secl2 eqs secl1) =
    (if repPO Os (Lsfm M secl2) (Lsfm M secl1) then U(:'v)
    else {} ) ∩
    if repPO Os (Lsfm M secl1) (Lsfm 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 {}

```

## 13.8 C2 Inference Rules in HOL

Recall in Section 13.3 that  $\mathcal{M} \models \phi$  denoted  $\mathcal{E}_{\mathcal{M}}[\llbracket \phi \rrbracket] = W$ , i.e.,  $\phi$  is true for all worlds in  $\mathcal{M}$ . Inference rules in the C2 calculus are sound because whenever  $\mathcal{M}$  satisfies all the hypotheses  $H_1 \cdots H_k$ , then  $\mathcal{M}$  satisfies conclusion  $C$  as well.

In our HOL implementation, we say Kripke structure  $M$  with partial orders  $O_i$  and  $O_s$  on integrity and security labels, respectively, satisfies an access-control logic formula  $f$  whenever the HOL semantic function `Efn`, whose definition appears in Section 13.7, applied to  $M$ ,  $O_i$ ,  $O_s$ , and  $f$  equals the universe of worlds in  $M$ . The definition of `sat` in HOL is as follows.

```

[sat_def]
⊢ ∀ M Oi Os f. (M, Oi, Os) sat f ⇔ (Efn Oi Os M f = U(:'world))

```

An inference rule in the C2 calculus of the form

$$\frac{H_1 \cdots H_k}{C}$$

has a corresponding theorem in HOL

$$\vdash \forall M O_i O_s. (M, O_i, O_s) \text{ sat } H_1 \Rightarrow \cdots \Rightarrow (M, O_i, O_s) \text{ sat } H_k \Rightarrow (M, O_i, O_s) \text{ sat } C,$$

where  $\Rightarrow$  corresponds to logical implication in HOL. Figures 13.7 and 13.8 show the HOL theorems corresponding to the C2 inference rules in Figure 13.3.

## 13.9 Using the Access Control Logic in HOL

### 13.9.1 Specifying Directory Paths for HOL

The implementation of the access-control logic in HOL should be located in a separate subdirectory from your own files. To enable HOL to find the access-control logic (ACL) theory files, we need to specify the path (either full path descriptions or the relative path from the working subdirectory) to the theory files.

**Figure 13.7** HOL Theorems Corresponding to C2 Calculus Inference Rules (1 of 2)[\[Controls\\_Eq\]](#)

$$\vdash \forall M \ Oi \ Os \ P \ f. \\ (M, Oi, Os) \text{ sat } P \text{ controls } f \iff (M, Oi, Os) \text{ sat } P \text{ says } f \text{ impf } f$$
[\[Reps\\_Eq\]](#)

$$\vdash \forall M \ Oi \ Os \ P \ Q \ f. \\ (M, Oi, Os) \text{ sat reps } P \ Q \ f \iff \\ (M, Oi, Os) \text{ sat } P \text{ quoting } Q \text{ says } f \text{ impf } Q \text{ says } f$$
[\[Modus Ponens\]](#)

$$\vdash \forall M \ Oi \ Os \ f_1 \ f_2. \\ (M, Oi, Os) \text{ sat } f_1 \Rightarrow \\ (M, Oi, Os) \text{ sat } f_1 \text{ impf } f_2 \Rightarrow \\ (M, Oi, Os) \text{ sat } f_2$$
[\[Says\]](#)

$$\vdash \forall M \ Oi \ Os \ P \ f. \ (M, Oi, Os) \text{ sat } f \Rightarrow (M, Oi, Os) \text{ sat } P \text{ says } f$$
[\[Controls\]](#)

$$\vdash \forall M \ Oi \ Os \ P \ f. \\ (M, Oi, Os) \text{ sat } P \text{ says } f \Rightarrow \\ (M, Oi, Os) \text{ sat } P \text{ controls } f \Rightarrow \\ (M, Oi, Os) \text{ sat } f$$
[\[Derived\\_Speaks\\_For\]](#)

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

Find the location of the ACL theory files on your machine. For illustrative purposes, the path we use here is  $\$ \{HOME\} / \text{Documents} / \text{RESEARCH} / \text{HOL} / \text{ACL} /$ . Do the following.

1. In your working directory, either locate or create the file *Holmakefile*.
2. Verify or add the path to the subdirectory containing the ACL theory files. This path is assigned to the variable INCLUDES.

```
INCLUDES=${HOME}/Documents/RESEARCH/HOL/ACL/
```

Other paths to other subdirectories may be included, where the path specifications are separated by a space.

### 13.9.2 Documentation for the ACL Implementation in HOL

The implementation of the access-control logic with its associated forward inference rules and tactics is fully described in the *Manual* subdirectory within the *HOL/ACL* subdirectory containing the ACL-HOL theory

**Figure 13.8** HOL Theorems Corresponding to C2 Calculus Inference Rules (2 of 2)[\[Reps\]](#)

$$\begin{aligned} &\vdash \forall M \ Oi \ Os \ P \ Q \ f. \\ &\quad (M, Oi, Os) \text{ sat } \text{reps } P \ Q \ f \Rightarrow \\ &\quad (M, Oi, Os) \text{ sat } P \text{ quoting } Q \text{ says } f \Rightarrow \\ &\quad (M, Oi, Os) \text{ sat } Q \text{ controls } f \Rightarrow \\ &\quad (M, Oi, Os) \text{ sat } f \end{aligned}$$
[\[And\\_Says\\_Eq\]](#)

$$\begin{aligned} &\vdash (M, Oi, Os) \text{ sat } P \text{ meet } Q \text{ says } f \iff \\ &\quad (M, Oi, Os) \text{ sat } P \text{ says } f \text{ andf } Q \text{ says } f \end{aligned}$$
[\[Quoting\\_Eq\]](#)

$$\begin{aligned} &\vdash \forall M \ Oi \ Os \ P \ Q \ f. \\ &\quad (M, Oi, Os) \text{ sat } P \text{ quoting } Q \text{ says } f \iff \\ &\quad (M, Oi, Os) \text{ sat } P \text{ says } Q \text{ says } f \end{aligned}$$
[\[Idemp\\_Speaks\\_For\]](#)

$$\vdash \forall M \ Oi \ Os \ P. \ (M, Oi, Os) \text{ sat } P \text{ speaks\_for } P$$
[\[Mono\\_Speaks\\_For\]](#)

$$\begin{aligned} &\vdash \forall M \ Oi \ Os \ P \ P' \ Q \ Q'. \\ &\quad (M, Oi, Os) \text{ sat } P \text{ speaks\_for } P' \Rightarrow \\ &\quad (M, Oi, Os) \text{ sat } Q \text{ speaks\_for } Q' \Rightarrow \\ &\quad (M, Oi, Os) \text{ sat } P \text{ quoting } Q \text{ speaks\_for } P' \text{ quoting } Q' \end{aligned}$$

files. Look for the file *aclHOLManual.pdf*.

The documentation of ACL inference rules and tactics follows the standard format used in the HOL reference manual.

1. In the box, the name of the function is on the left, and the file containing its definition is on the right.
2. The ML type signature of the function immediately follows the boxed information.
3. **Synopsis** is describes the purpose of the function.
4. **Description** shows how the function is used, typically presented as a rule.
5. **Failure** states the conditions when the function will not work.
6. **Example** shows an example in HOL using the function.
7. **Implementation** reveals the ML code implementing the function.
8. **See also** is a pointer to related functions.



### 13.9.3 The ACL\_ASSUM and CONTROLS Inference Rules

The first two ACL inference rules we introduce in HOL are the `ACL_ASSUM` and `CONTROLS` rules. Their documentation, excerpted from the the manual *An Access-Control Logic in HOL*, follows. In the next section, we show how they are used to do ACL proofs in HOL.

#### ACL\_ASSUM Inference Rule

The following is from the manual *An Access-Control Logic in HOL*. It describes the *ACL\_ASSUM* inference rule, which is similar in function to the HOL inference rule *ASSUME*, except that it is applied to a formula  $f$  in the access-control logic, and returns a theorem of the form  $[(M, Oi, Os) \text{ sat } f] \vdash (M, Oi, Os) \text{ sat } f$ .

<code>ACL_ASSUM</code>	<code>(acl_infRules)</code>
------------------------	-----------------------------

`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

$(M, Oi, Os) \text{ sat } f \vdash (M, Oi, Os) \text{ sat } f$ .

```

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

```

#### Failure

Fails unless  $f$  has type `Form`.

#### Example

The following application:

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

produces the following result:

<pre> val a1 =  [.]  - (M,Oi,Os) sat Token says Role says f : thm </pre>
--

## 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

## Controls Inference Rule

CONTROLS

(acl\_infRules)

CONTROLS : thm->thm -> thm

## Synopsis

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

## Description

$$\begin{array}{l}
 A1 \mid- (M, Oi, Os) \text{ sat } P \text{ controls } f \\
 A2 \mid- (M, Oi, Os) \text{ sat } P \text{ says } f \\
 \hline
 A1 \text{ u } A2 \mid- (M, Oi, Os) \text{ sat } f
 \end{array}
 \quad \text{CONTROLS}$$

## 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

### 13.9.4 Using the ACL\_ASSUM and Controls Inference Rules

In this section, we develop a theory *example1Theory*. We will build the theory through a series of examples.

#### Example 13.3

In this example, we build the basic algebraic types for commands and principals. The theory we create is *example1Theory*. Do the following.

1. Create, as normal, the Script file *example1Script.sml* that builds *example1Theory*.
2. **Important:** In the file *Holmakefile*, assign the paths to the ACL subdirectory containing your HOL-compiled access-control logic theory files.
3. Within *example1Script.sml*, add the typical infrastructure, i.e., the `structure`, `new_theory`, `print_theory`, and `export_theory` statements.
4. **Important:** in addition to the *HolKernel*, *boolLib*, *Parse*, and *bossLib* theories and structures required by *Holmake*, **open** the structure *acl\_infRules*. These are the ML functions that are the access-control logic inference rules implemented in HOL. The functions are located in the subdirectory containing the ACL-HOL theory files.
5. Add to your script file the HOL commands defining the following datatypes.

```

commands = go | nogo | launch | abort
staff    = Alice | Bob | Carol | Dan

```

6. In a terminal, execute *Holmake* to make sure your script file compiles as expected.

◇

**Example 13.4**

In this example, we extend *example1Theory* by proving and saving the HOL theorem *example1Theorem* below. The theorem corresponds to a derived inference rule in the access-control logic.

[*example1Theorem*]

$$\begin{aligned} \vdash (M, Oi, Os) \text{ sat Name Alice says prop go} &\Rightarrow \\ (M, Oi, Os) \text{ sat Name Alice controls prop go} &\Rightarrow \\ (M, Oi, Os) \text{ sat prop go} \end{aligned}$$

The “pencil and paper” proof in the access-control logic is as follows.

1.	<i>Alice says</i> $\langle go \rangle$	Assumption
2.	<i>Alice controls</i> $\langle go \rangle$	Assumption
3.	$\langle go \rangle$	2, 1 Controls

The HOL theorem *example1Theorem* corresponds to the derived inference rule

$$\frac{\text{Alice controls } \langle go \rangle \quad \text{Alice says } \langle go \rangle}{\langle go \rangle},$$

where the two hypotheses correspond to the two antecedents of the theorem, and the conclusion of the implication in HOL corresponds to the conclusion of the derived inference rule.

Our “pencil and paper” proof above shows us the steps we need to do in HOL. We need to:

1. Introduce HOL theorems corresponding to the access-control logic assumptions. We use `ACL_ASSUM` to do this.
2. Apply the HOL inference rule corresponding to the *Controls* inference rule. We use the `CONTROLS` inference rule to do this.
3. Similar to all the forward proofs we did previously in HOL, we need to discharge the assumptions from the assumption list so they appear as antecedents of an implication in the conclusion of the final theorem. We use the `DISCH` inference rule to do this.

We now do our first proof using the ACL theories. Do the following:

1. Open *example1Script.sml*, start a HOL session, load and open *example1Theory* and *acl\_infRules*.
2. **Important:** using the *Printing switches* menu under the *HOL* menu button, turn on types and and turn off unicode. We will want to see types explicitly initially to understand what is happening.
3. Add the following code to *example1Script.sml* just before the `print_theory` and `export_theory` commands.

```
val th1 = ACLASSUM'((Name Alice) says (prop go)):(commands,staff,'d','e')Form'<';
```

4. Execute the above in the HOL interpreter. The result is shown below.

```

- val th1 = ACL_ASSUM``((Name Alice) says (prop go)):(commands,staff,'d','e)Form``;
> val th1 =
[.]
|- ((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po),
   (Os :'e po)) sat
   Name Alice says (prop go :(commands, staff, 'd, 'e) Form)
   : thm

```

1

Looking at the above result, we observe the following.

1. The access-control logic formula

```((Name Alice) says (prop go)):(commands,staff,'d,'e)Form``,`

which we provided as an assumption is explicitly typed as ```:(commands,staff,'d,'e)Form```.

2. Recall from the syntax definition of ACL formulas in HOL defined in Section 13.5, that in general Kripke structures  $M$  have type `('a, 'b, 'c, 'd, 'e)Kripke`, where the type variables are associated with the following elements:

- (a) `'a` is the type of *atomic propositions*.
- (b) `'b` is the type of worlds in the Kripke structure  $M$ .
- (c) `'c` is the type of *principals*.
- (d) `'d` is the type of *integrity labels*, which are partially ordered by the relation `Oi`.
- (e) `'e` is the type of *security labels*, which are partially ordered by the relation `Os`.

3. Recall from the syntax definition of ACL formulas in HOL defined in Section 13.5, that in general the type parameters of ACL *formulas* are `'a`, `'c`, `'d`, and `'e`, e.g.,

```f:( 'a, 'c, 'd, 'e)Form```.

Recall that the set of worlds in Kripke structure  $M$  does not appear explicitly in ACL formulas. Hence, their type `'b` is not included as a type parameter of `Form`.

4. The definition of `ACL_ASSUM` always assigns the type of worlds to be `: 'b`. While you will rarely, if ever, need to worry about the explicit type of worlds, you should avoid using the type variable `'b` in any of your ACL formulas to avoid confusing HOL.

Continuing on with our proof, do the following.

1. Add the following code to *example1Script.sml*.

```

val th2 = ACLASSUM`((Name Alice) controls (prop go)):(commands,staff,'d,'e)Form`
val th3 = CONTROLS th2 th1;
val th4 = DISCH(hd(hyp th2)) th3;
val th5 = DISCH(hd(hyp th1)) th4;

```

2. Execute each line separately in the HOL interpreter and see what happens.
3. What you should see is shown below.

```

- val th2 = ACL_ASSUM``((Name Alice) controls (prop go)):(commands,staff,'d,'e)Form``;
> val th2 =
  [...]
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po),
      (Os :'e po)) sat
      Name Alice controls (prop go :(commands, staff, 'd, 'e) Form)
      : thm
- val th3 = CONTROLS th2 th1;
> val th3 =
  [...]
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po),
      (Os :'e po)) sat (prop go :(commands, staff, 'd, 'e) Form)
      : thm
- val th4 = DISCH(hd(hyp th2)) th3;
> val th4 =
  [...]
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po),
      (Os :'e po)) sat
      Name Alice controls (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)
      : thm
- val th5 = DISCH(hd(hyp th1)) th4;
> val th5 =
  [...]
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po),
      (Os :'e po)) sat
      Name Alice says (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      Name Alice controls (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)
      : thm

```

Pay particular attention to the application of the `CONTROLS` inference rule used to calculate the value of `th3`. The theorems `th4` and `th5` are computed by discharging the assumptions in the assumption list using `DISCH` to discharge the assumptions in the desired order.

As normal, we package up the entire proof within a `let` expression and save the resulting theorem. Do the following:

1. Add the following code to your file.

```

(* Package up the proof into a single function *)
val example1Theorem =
let
  val th1 = ACLASSUM``((Name Alice) says (prop go)):(commands,staff,'d,'e)Form``
  val th2 = ACLASSUM
    ``((Name Alice) controls (prop go)):(commands,staff,'d,'e)Form``
  val th3 = CONTROLS th2 th1
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;

(* We save the theorem by using save_thm *)
val _ = save_thm("example1Theorem",example1Theorem)

```

2. Executing the above yields the result shown below.

```

- (* Package up the proof into a single function *)
val example1Theorem =
let
  val th1 = ACL_ASSUM``(Name Alice) says (prop go):(commands,staff,'d','e')Form``
  val th2 = ACL_ASSUM``(Name Alice) controls (prop go):(commands,staff,'d','e')Form``
  val th3 = CONTROLS th2 th1
  val th4 = DISCH(hd(hyp th2)) th3
in
  DISCH(hd(hyp th1)) th4
end;

(* We save the theorem by using save_thm *)
val _ = save_thm("example1Theorem",example1Theorem);
> val example1Theorem =
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),
      (Os :'e po)) sat
      Name Alice says (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      Name Alice controls (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)
: thm

```

Finally, make sure you save the file and execute Holmake in a terminal in the same subdirectory as `example1Script.sml`. ◇

### 13.9.5 Goal-Oriented Access-Control Logic Proofs

Using the forward inference rules of the access-control logic corresponds most closely with the proofs shown in [Chin and Older, 2010]. Goal-oriented proofs of ACL theorems is often a combination of using tactics<sup>1</sup> corresponding to the ACL inference rules and/or using `PROVE_TAC` with ACL theorems corresponding to the inference rules shown in Figure 13.3 and in [Chin and Older, 2010]. We illustrate both techniques in the following examples.

#### Example 13.5

We prove *example1Theorem* again using `PROVE_TAC`. Do the following.

1. Open the file *example1Script.sml* in Emacs and start a HOL session.
2. Modify your file to **open** two theories in addition to *acl\_infRules*: *aclrulesTheory* and *aclDrulesTheory*.
3. Load and open *acl\_infRules*, *aclrulesTheory*, and *aclDrulesTheory*.
4. Add the following code to your file.

```

set_goal([],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat Name Alice controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``)

```

Notice that the Kripke structure *M* is explicitly typed.

<sup>1</sup>Thanks to Steven Perkins for developing the access-control logic tactics

5. Execute the above code within the HOL interpreter. For complete clarity, make sure you enable types to be printed explicitly and turn off unicode. The result you should see is below.

4

```

- set_goal([],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po), (Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat Name Alice controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``);
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

    ((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po),
      (Os :'e po)) sat
    Name Alice says (prop go :(commands, staff, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    Name Alice controls (prop go :(commands, staff, 'd, 'e) Form) ==>
    (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)

    : proofs

```

In the forward version of the proof, we used the inference rule `CONTROLS`. Examining the documentation for `CONTROLS` in Section 13.9.3, we see that the inference rule uses the *Controls* inference rule, which is proved in *aclDrulesTheory*. The theorem is shown below.

[Controls]

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

We use the *Controls* theorem as part of `PROVE_TAC` to prove the goal. Do the following.

1. Add the following code to your file.

`PROVE_TAC[Controls]`

2. Execute the tactic in the HOL interpreter. The result is shown below.

5

```

- e(PROVE_TAC[Controls]);
OK..
Meson search level: ....
> val it =
  Initial goal proved.
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po),
      (Os :'e po)) sat
      Name Alice says (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      Name Alice controls (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)

  : proof

```

As usual, we replace the interactive proof using `set_goal` with `TAC_PROOF`. We name the theorem `example1TheoremA`. Do the following.



1. Replace the code using `set_goal` and `PROVE_TAC` with the following.

```

val example1TheoremA =
TAC.PROOF([],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat Name Alice controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``),
PROVE_TAC[Controls])

val _ = save_thm("example1TheoremA",example1TheoremA)

```

2. What you should see appears below.

6

```

- val example1TheoremA =
TAC_PROOF([],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat Name Alice controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``),
PROVE_TAC[Controls])

val _ = save_thm("example1TheoremA",example1TheoremA);
Meson search level: ....
> val example1TheoremA =
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),
    (Os :'e po)) sat
    Name Alice says (prop go :(commands, staff, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    Name Alice controls (prop go :(commands, staff, 'd, 'e) Form) ==>
    (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)
: thm

```

As usual, save the contents of *example1Script.sml*, and run Holmake in a terminal in the subdirectory containing *example1Script.sml*. ◇

### The Tactic ACL\_CONTROLS\_TAC

Sometimes, our goal-oriented proofs cannot be proved solely using `PROVE_TAC`. In these cases, we have tactics corresponding to the forward inference rules. As an example, we show how `ACL_CONTROLS_TAC` is used.

The documentation from the ACL-HOL manual for `ACL_CONTROLS_TAC` is shown below. Its format is the same as the format for the inference rules `ACL_ASSUM` and `CONTROLS`.

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  $A \text{ ?- } (M, Oi, Os) \text{ sat } f$ , returns a two new subgoals in the form  $A \text{ ?- } (M, Oi, Os) \text{ sat } p \text{ controls } f$  and  $A \text{ ?- } (M, Oi, Os) \text{ sat } p \text{ says } f$ .

```

      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

### Example 13.6

In this example, we reprove the theorem *example1Theorem* a third time using the `ACL_CONTROLS_TAC` tactic, which is described above. Do the following.

1. Open *example1Script.sml* in Emacs and start a HOL process.
2. Put the following code into your file. Make sure you turn on the HOL printing switches to enable types to be shown; turn off unicode.

```

set_goal([],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat Name Alice controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``)

```

3. Execute the above in HOL. You should see what appears below.

```

- set_goal([],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat Name Alice controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``);
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

      ((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),
        (Os :'e po)) sat
      Name Alice says (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      Name Alice controls (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)

: proofs

```

7

We simplify the goal as much as possible using the tactic `REPEAT STRIP_TAC`. Do the following.

1. Add the following code to your file.

```
REPEAT STRIP_TAC THEN
ACL_CONTROLS_TAC ``Name Alice `` THEN
PROVE_TAC[]
```

2. Execute each tactic alone to see its effect.

3. What you should see is shown below.

```

- e(REPEAT_STRIP_TAC);
OK..
1 subgoal:
> val it =

  ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po), (Os : 'e po)) sat
  (prop go : (commands, staff, 'd, 'e) Form)
  -----
  0. ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po),
      (Os : 'e po)) sat
      Name Alice says (prop go : (commands, staff, 'd, 'e) Form)
  1. ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po),
      (Os : 'e po)) sat
      Name Alice controls (prop go : (commands, staff, 'd, 'e) Form)
  : proof
- e(ACL_CONTROLS_TAC ``Name Alice``);
OK..
2 subgoals:
> val it =

  ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po), (Os : 'e po)) sat
  Name Alice says (prop go : (commands, staff, 'd, 'e) Form)
  -----
  0. ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po),
      (Os : 'e po)) sat
      Name Alice says (prop go : (commands, staff, 'd, 'e) Form)
  1. ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po),
      (Os : 'e po)) sat
      Name Alice controls (prop go : (commands, staff, 'd, 'e) Form)

  ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po), (Os : 'e po)) sat
  Name Alice controls (prop go : (commands, staff, 'd, 'e) Form)
  -----
  0. ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po),
      (Os : 'e po)) sat
      Name Alice says (prop go : (commands, staff, 'd, 'e) Form)
  1. ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po),
      (Os : 'e po)) sat
      Name Alice controls (prop go : (commands, staff, 'd, 'e) Form)

  2 subgoals
  : proof
- e(PROVE_TAC[]);
OK..
Meson search level: ..

Goal proved.
[.]
|- ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po),
    (Os : 'e po)) sat
    Name Alice controls (prop go : (commands, staff, 'd, 'e) Form)

.... output omitted ....
- e(PROVE_TAC[]);
OK..
Meson search level: ..

.... output omitted ....

> val it =
  Initial goal proved.
  |- ((M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po),
      (Os : 'e po)) sat
      Name Alice says (prop go : (commands, staff, 'd, 'e) Form) ==>
      (M, Oi, Os) sat
      Name Alice controls (prop go : (commands, staff, 'd, 'e) Form) ==>
      (M, Oi, Os) sat (prop go : (commands, staff, 'd, 'e) Form)
  : proof

```

Notice that application of `ACL_CONTROLS_TAC ``Name Alice``` results in two subgoals, one corresponding to `Name Alice says (prop go)` and the other `Name Alice controls (prop go)`, which corresponds to the *Controls* inference rule.

As usual, we replace the above code using `set_goal`, with `TAC_PROOF`. Do the following.

1. Replace all the code starting with `set_goal` with the following.

```
val example1TheoremB =
TAC_PROOF([],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat Name Alice controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``),
  REPEAT STRIP_TAC THEN
  ACL_CONTROLS_TAC ``Name Alice`` THEN
  PROVE_TAC[])

val _ = save_thm("example1TheoremB",example1TheoremB)
```

2. Executing the above within HOL produces the following.

```
- val example1TheoremB =
TAC_PROOF([],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),(Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat Name Alice controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``),
  REPEAT STRIP_TAC THEN
  ACL_CONTROLS_TAC ``Name Alice`` THEN
  PROVE_TAC[])

val _ = save_thm("example1TheoremB",example1TheoremB);
Meson search level: ..
Meson search level: ..
> val example1TheoremB =
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke),(Oi :'d po),
      (Os :'e po)) sat
      Name Alice says (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      Name Alice controls (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)
  : thm
```

As before, save the contents of *example1Script.sml* and execute `Holmake` in a terminal within the subdirectory containing *example1Script.sml*. ◇

### 13.9.6 More Inference Rules and Tactics

We introduce the `SPEAKS_FOR` inference rule. Its description follows and is taken from the ACL-HOL manual.

**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  $P \text{ speaks\_for } Q$ , the second is  $P \text{ says } f$ , 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

In the following examples, we prove the same theorem three times. The first time using forward inference rules. The second time using only PROVE\_TAC. The third time using a combination of ACL tactics and manipulating the assumptions using forward inference rules.

### Example 13.7

We prove the theorem

[\[example2Theorem\]](#)

```

⊢ (M, Oi, Os) sat Name Alice says prop go ⇒
  (M, Oi, Os) sat Name Alice speaks_for Name Bob ⇒
  (M, Oi, Os) sat Name Bob controls prop go ⇒
  (M, Oi, Os) sat prop go

```

The “pencil and paper” proof in the access-control logic is as follows.

1.	<i>Alice</i> says $\langle go \rangle$	Assumption
2.	<i>Alice</i> $\Rightarrow$ <i>Bob</i>	Assumption
3.	<i>Bob</i> controls $\langle go \rangle$	Assumption
4.	<i>Bob</i> says $\langle go \rangle$	2, 1 Speaks For
5.	$\langle go \rangle$	3, 4 Controls

The forward proof in HOL mirrors the “pencil and paper” proof above. Do the following.

1. Open the file *example1Script.sml* and add the following code.

```
(* Package up the proof into a single function *)
val example2Theorem =
let
  val th1 =
    ACLASSUM' '((Name Alice) says (prop go)):(commands, staff, 'd, 'e)Form' '
  val th2 =
    ACLASSUM' '((Name Alice) speaks_for (Name Bob)):(commands, staff, 'd, 'e)Form' '
  val th3 =
    ACLASSUM' '((Name Bob) controls (prop go)):(commands, staff, 'd, 'e)Form' '
  val th4 = SPEAKS_FOR th2 th1
  val th5 = CONTROLS th3 th4
  val th6 = DISCH(hd(hyp th3)) th5
  val th7 = DISCH(hd(hyp th2)) th6
in
  DISCH(hd(hyp th1)) th7
end;

(* We save the theorem by using save_thm *)
val _ = save_thm("example2Theorem", example2Theorem)
```

2. Execute the above within HOL.
3. What you should see appears below.



10

```

- (* Package up the proof into a single function *)
val example2Theorem =
let
  val th1 =
    ACL_ASSUM``(Name Alice) says (prop go):(commands,staff,'d,'e)Form``
  val th2 =
    ACL_ASSUM``(Name Alice) speaks_for (Name Bob):(commands,staff,'d,'e)Form``
  val th3 =
    ACL_ASSUM``(Name Bob) controls (prop go):(commands,staff,'d,'e)Form``
  val th4 = SPEAKS_FOR th2 th1
  val th5 = CONTROLS th3 th4
  val th6 = DISCH(hd(hyp th3)) th5
  val th7 = DISCH(hd(hyp th2)) th6
in
  DISCH(hd(hyp th1)) th7
end;

(* We save the theorem by using save_thm *)
val _ = save_thm("example2Theorem",example2Theorem);
> val example2Theorem =
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po),
      (Os :'e po)) sat
      Name Alice says (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      ((Name Alice speaks_for Name Bob)
       :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      Name Bob controls (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)
: thm

```

As always, execute Holmake in the subdirectory containing *example1Script.sml* in a terminal to make sure all compiles as expected. ◇

### Example 13.8

We prove the above theorem again, except that this time we use only PROVE\_TAC with the ACL theorems *Derived\_Speaks\_For* and *Controls*. Both theorems are proved in *aclDrulesTheory*. Both theorems are the basis for the SPEAKS\_FOR and CONTROLS inference rules, respectively.

Do the following.

1. Open the file *example1Script.sml* and add the following code.

```

val example2TheoremA =
TAC.PROOF([],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po), (Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat (Name Alice speaks_for Name Bob) ==>
    (M,Oi,Os) sat Name Bob controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``),
PROVE_TAC[Derived_Speaks_For, Controls])

val _ = save_thm("example2TheoremA",example2TheoremA)

```

2. Execute the above in HOL.
3. What you should see appears below.

```

- val example2TheoremA =
TAC_PROOF([[]],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po), (Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat (Name Alice speaks_for Name Bob) ==>
    (M,Oi,Os) sat Name Bob controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``),
PROVE_TAC[Derived_Speaks_For, Controls])

val _ = save_thm("example2TheoremA", example2TheoremA);
Meson search level: .....
> val example2TheoremA =
  |- ((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po),
      (Os :'e po)) sat
      Name Alice says (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      ((Name Alice speaks_for Name Bob)
       :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      Name Bob controls (prop go :(commands, staff, 'd, 'e) Form) ==>
      (M,Oi,Os) sat (prop go :(commands, staff, 'd, 'e) Form)
  : thm

```



### Example 13.9

Finally, we prove the same theorem a third time. This time we show how we use PAT\_ASSUM in conjunction with the forward inference rules to manipulate the assumptions on the assumption list. Do the following.

1. Open the file *example1Script.sml* and add the following code.

```

val example2TheoremB =
TAC_PROOF([[]],
  ``((M :(commands, 'b, staff, 'd, 'e) Kripke), (Oi :'d po), (Os :'e po)) sat
    Name Alice says (prop go) ==>
    (M,Oi,Os) sat (Name Alice speaks_for Name Bob) ==>
    (M,Oi,Os) sat Name Bob controls (prop go) ==>
    (M,Oi,Os) sat (prop go)``),
REPEAT STRIP_TAC THEN
ACL.CONTROLS_TAC ``Name Bob`` THEN
ASM.REWRITE_TAC[] THEN
PAT_ASSUM
  ``((M,Oi,Os) sat (Name Alice speaks_for Name Bob))``
(fn th1 =>
  (PAT_ASSUM
    ``((M,Oi,Os) sat (Name Alice says (prop go)))``
    (fn th2 => ASSUME_TAC(SPEAKS_FOR th1 th2)))) THEN
PROVE_TAC[]

```

2. Execute the above in HOL.
3. What you should see is below. Notice how the SPEAKS\_FOR forward inference rule is used within PAT\_ASSUM to do a small forward proof in the assumption list.

12

```

- val example2TheoremB =
TAC_PROOF([[],
  `` (M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po), (Os : 'e po)) sat
    Name Alice says (prop go) ==>
    (M, Oi, Os) sat (Name Alice speaks_for Name Bob) ==>
    (M, Oi, Os) sat Name Bob controls (prop go) ==>
    (M, Oi, Os) sat (prop go) ``),
REPEAT STRIP_TAC THEN
ACL_CONTROLS_TAC ``Name Bob`` THEN
ASM_REWRITE_TAC[] THEN
PAT_ASSUME
  `` (M, Oi, Os) sat (Name Alice speaks_for Name Bob) ``
  (fn th1 =>
    (PAT_ASSUME
      `` (M, Oi, Os) sat (Name Alice says (prop go)) ``
      (fn th2 => ASSUME_TAC (SPEAKS_FOR th1 th2)))) THEN
PROVE_TAC[]

val _ = save_thm("example2TheoremB", example2TheoremB);

<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
<<HOL message: inventing new type variable names: 'a, 'b, 'c>>
Meson search level: ..
> val example2TheoremB =
  |- (M : (commands, 'b, staff, 'd, 'e) Kripke), (Oi : 'd po),
    (Os : 'e po)) sat
    Name Alice says (prop go : (commands, staff, 'd, 'e) Form) ==>
    (M, Oi, Os) sat
    ((Name Alice speaks_for Name Bob)
      : (commands, staff, 'd, 'e) Form) ==>
    (M, Oi, Os) sat
    Name Bob controls (prop go : (commands, staff, 'd, 'e) Form) ==>
    (M, Oi, Os) sat (prop go : (commands, staff, 'd, 'e) Form)
  : thm

```

◇

## 13.10 Exercises

**Exercise 13.10.1** Build a theory `solutions1Theory`, which has `example1Theory` as a parent theory.

- A. Do a **forward proof** of the following inference rule in HOL using the data type definitions in `example1Theory` for `commands` and `staff`.

$$\frac{\text{Alice says } \langle go \rangle \quad \text{Bob says } \langle go \rangle}{\text{Alice \& Bob says } \langle go \rangle}$$

Package up your entire proof into a single function using `let` statements, as usual. Make sure your theorem has an empty list of assumptions and that the antecedents of the conclusion appear in the same order as the assumptions in the derived inference rule above. That is, the HOL theorem corresponding to the derived rule in the access control logic  $\frac{a}{c}b$  is  $\{\} \vdash a \implies b \implies c$  in HOL.

**Hints:** (1) make sure you explicitly give the type signature of the ACL formulas to which you apply `ACL_ASSUME`, (2) turn on types so they are explicitly shown, and (3) check the ACL-HOL manual for relevant inference rules, e.g., `ACL_CONJ`, and `AND_SAYS_RL`.

Name your theorem `aclExercisel`. The theorem you should prove is below.

`[aclExercisel]`

```

  ⊢ (M, Oi, Os) sat Name Alice says prop go ⇒
    (M, Oi, Os) sat Name Bob says prop go ⇒
    (M, Oi, Os) sat Name Alice meet Name Bob says prop go

```

- B. Do a **goal-oriented proof** of the same theorem in part A using only **PROVE\_TAC** applied to the appropriate theorems, e.g., Controls. Name the theorem `aclExercise1A`. The theorem you should prove using tactics is

```
[aclExercise1A]
```

```

  ⊢ (M, Oi, Os) sat Name Alice says prop go ⇒
    (M, Oi, Os) sat Name Bob says prop go ⇒
    (M, Oi, Os) sat Name Alice meet Name Bob says prop go

```

- C. Do a **goal-oriented proof** of the same theorem in part A using the ACL tactics `ACL_AND_SAYS_RL_TAC` and `ACL_CONJ_TAC`. You may use `PROVE_TAC` with **an empty list of theorems** at the end to finish the proof.

Name your theorem `aclExercise1B`.

```
[aclExercise1B]
```

```

  ⊢ (M, Oi, Os) sat Name Alice says prop go ⇒
    (M, Oi, Os) sat Name Bob says prop go ⇒
    (M, Oi, Os) sat Name Alice meet Name Bob says prop go

```

**Exercise 13.10.2** Extend the theory `solutions1Theory` defined in the previous exercise by proving the theorems listed below.

- A. Do a **forward proof** of the following inference rule in HOL.

$$\frac{\text{Alice says } \langle go \rangle \quad \text{Alice controls } \langle go \rangle \quad \langle go \rangle \supset \langle launch \rangle}{\text{Bob says } \langle launch \rangle}$$

Package up your entire proof into a single function using `let` statements, as usual. Make sure your theorem has an empty list of assumptions and that the antecedents of the conclusion appear in the same order as the assumptions in the derived inference rule above.

Name your theorem `aclExercise2`. The theorem you should prove is below.

```
[aclExercise2]
```

```

  ⊢ (M, Oi, Os) sat Name Alice says prop go ⇒
    (M, Oi, Os) sat Name Alice controls prop go ⇒
    (M, Oi, Os) sat prop go impf prop launch ⇒
    (M, Oi, Os) sat Name Bob says prop launch

```

- B. Do a **goal-oriented proof** of the same theorem in part A using only **PROVE\_TAC** applied to the appropriate theorems. Name the theorem `aclExercise2A`. The theorem you should prove using `PROVE_TAC` alone is

```
[aclExercise2A]
```

```

  ⊢ (M, Oi, Os) sat Name Alice says prop go ⇒
    (M, Oi, Os) sat Name Alice controls prop go ⇒
    (M, Oi, Os) sat prop go impf prop launch ⇒
    (M, Oi, Os) sat Name Bob says prop launch

```

**Hint:** You may want to load and open `aclrulesTheory` and `aclDrulesTheory` to make use of the theorems `Modus_Ponens`, `Controls`, and `Says`.

- C. Do a **goal-oriented proof** of the same theorem in Part A. Do the proof by first doing `REPEAT STRIP_TAC` followed by `ACL_SAYS_TAC`. Then use `PAT_ASSUME` with `ACL` forward inference rules on the assumptions to prove the goal. Name the theorem `aclExercise2B`.

`[aclExercise2B]`

```

⊢ (M, Oi, Os) sat Name Alice says prop go ⇒
  (M, Oi, Os) sat Name Alice controls prop go ⇒
  (M, Oi, Os) sat prop go impf prop launch ⇒
  (M, Oi, Os) sat Name Bob says prop launch

```

BLANK PAGE

## Concepts of Operation in the Access-Control Logic and HOL

---

### 14.1 Concepts of Operations

Users of systems, where systems are machines, software applications, protocols, or processes coordinating the work among human organizations, typically have behavioral models of the systems they use. These models are *concepts of operations* or CONOPS. As defined by IEEE Standard 1362 [IEE, 1998], a CONOPS expresses the “characteristics for a proposed system from a user’s perspective. A CONOPS also describes the user organization, mission, and objectives from an integrated systems point of view.”

The US military has a similar definition of CONOPS in Joint Publication 5-0, Joint Operational Planning [JP5, 2011]. For military leaders planning a mission, a CONOPS describes “how the actions of components and organizations are integrated, synchronized, and phased to accomplish the mission.”

Put more plainly, a CONOPS describes the who, what, when, and why. When we explicitly address security and integrity concerns, we state how we know with whom we are dealing and what authority they have, i.e., how we authenticate and authorize people, processes, statements, and commands.

Figure 14.1(a) shows a ladder diagram of a simple CONOPS. Here is its interpretation.

1. Time moves from top to bottom.
2. The vertical line labeled with *Operator* at the top is used to record the sequence of inputs and output to and from the *Operator* over time.
3. *Operator* receives an input  $\langle command_1 \rangle$ .
4. *Operator* transmits an output  $\langle command_2 \rangle$  after receiving  $\langle command_1 \rangle$ .

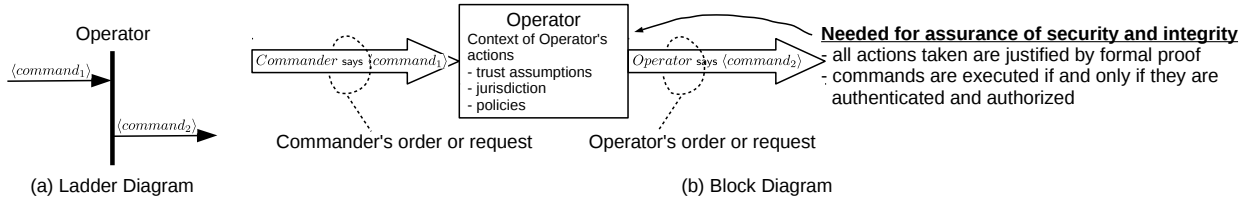
Ladder diagrams are used often to describe protocols in terms of the sequence of messages exchanged among principals in order to accomplish a specific task, such as, distributing cryptographic keys and establishing a secure communication session. However, ladder diagrams often do not specify the policies and context justifying the action taken by a principal after receiving a specified input.

In addition to ladder diagrams, block diagrams with more detail are used to describe a sequence of operations. Figure 14.1(b) shows a block diagram of a simple CONOPS. Here is its interpretation.

1. The flow of command and control in this figure is from left to right. The *Commander* issues a command by some means (speaking, writing, electronically, telepathy, etc.). This is symbolized by

*Commander* says  $\langle command_1 \rangle$ .

2. The box in the center labeled **Operator** shows the Operator receiving the Commander’s command on the left. Inside the box are the things the Operator “knows”, i.e., the context within which he attempts to justify acting on Alice’s command. The context might include a policy that if the Operator receives

**Figure 14.1** Flow of Command and Control (C2) for a Simple CONOPS

a particular command, such as *go*, then the Operator is to issue another command, such as *launch*. The Operator's actions to the Commander's command are based on the Operator's operational context, which includes statements or assumptions such as the Commander has the authority, jurisdiction, or is to be believed on matters related to the command the Commander has made.

3. The arrow coming from the right hand side of the box shows the Operator's statement or command, which is symbolized by

*Operator says*  $\langle \text{command}_2 \rangle$ .

4. What Figure 14.1 shows is one C2 sequence starting from left to right. The Operator gets an order from the Commander. The Operator decides based on the Commander's order and what is known (the statements inside the box), that it is a good idea to issue *command*<sub>2</sub>. This is symbolized by

*Operator says*  $\langle \text{command}_2 \rangle$ .

Regarding the comment in Figure 14.1, for assurance what we want is a logical justification of the actions the Operator takes given the received order and the operational context. For us, logical justifications are *proofs in access-control logic*.

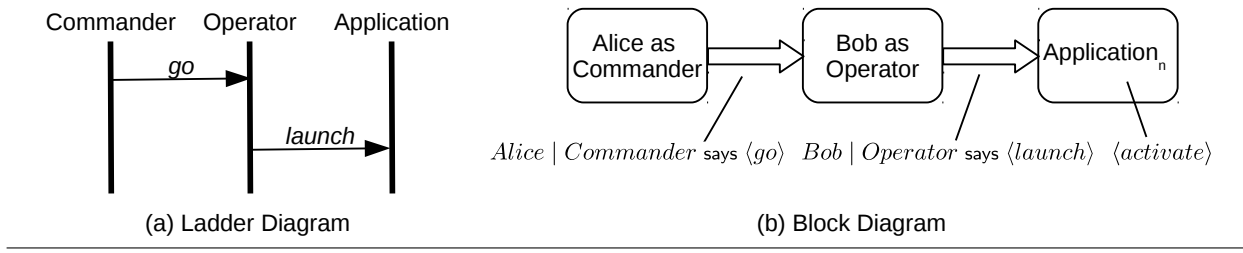
Security vulnerabilities often result from inconsistencies among CONOPS at various levels of abstraction. Military commanders might assume only authorized operators are able to launch an application, whereas the application itself might incorrectly trust that all orders it receives are from authorized operators and never authenticate the inputs it receives. Any *design for assurance* methodology must address authentication and authorization in order to *avoid vulnerabilities* due to unauthorized access or control. Rigorous assurance requires mathematical models and proofs. Our intent is illustrate a structured way to achieve security by design.

## 14.2 A Command and Control Example

Figure 14.2 illustrates a simple command and control scenario. The envisioned concept-of-operations is as follows.

1. *Alice* in the role of *Commander* gives the *go* command to Bob who functions as an *Operator*.
2. Bob recognizes that the *Commander* has the authority to give the *go* command. He also knows that *Alice* is the *Commander*. The operational policy given to *Operators* is when they receive a *go* command, they know that they should *launch* the application for which they are responsible. In Bob's case, it is *Application<sub>n</sub>*.



**Figure 14.2** A Command and Control Chain

3. When  $Application_n$  receives a *launch* command from an *Operator*, in this case *Bob*, it *activates*.

Our next task is to formalize the informal CONOPS description using the access-control logic. We take the following approach.

1. The *principals* are *people*, *roles*, and *applications*.
2. Command authority is associated with *roles*, *not people*. We model this in the access-control logic as

$$Role \text{ controls } \langle command \rangle$$

3. *People act in their assigned roles*. We model this as

$$Person \mid Role \text{ says } \langle command \rangle$$

4. Recognizing that a person is legitimately acting in an assigned role giving a command is modeled as a *delegation*.

$$Person \text{ reps } Role \text{ on } \langle command \rangle$$

Using this approach, the *Reps* inference in this case looks like

$$Reps \quad \frac{Role \text{ controls } \langle command \rangle \quad Person \text{ reps } Role \text{ on } \langle command \rangle \quad Person \mid Role \text{ says } \langle command \rangle}{\langle command \rangle}$$

Our objective is to derive sound inference rules in the access-control logic justifying the actions of *Bob* as *Operator* and  $Application_n$ . We do this first in the “pencil and paper” version of the access-control logic. Then we implement and check our proofs using HOL.

### Example 14.1

The derived inference rule justifying Bob as Operator issuing a *launch* command when he receives the *go* command from Commander Alice is

$$OpRule1 \quad \frac{\begin{array}{l} Commander \text{ controls } \langle go \rangle \quad Alice \text{ reps } Commander \text{ on } \langle go \rangle \\ Alice \mid Commander \text{ says } \langle go \rangle \quad \langle go \rangle \supset \langle launch \rangle \end{array}}{Bob \mid Operator \text{ says } \langle launch \rangle}$$

The proof of *OpRule1* is shown in Figure 14.3.

◇

**Figure 14.3** Proof Justifying Operator's Action

1.	<i>Commander</i> controls $\langle go \rangle$	Commander's Authority
2.	<i>Alice</i> reps <i>Commander</i> on $\langle go \rangle$	Alice is recognized as the Commander
3.	<i>Alice</i>   <i>Commander</i> says $\langle go \rangle$	Alice's order
4.	$\langle go \rangle \supset \langle launch \rangle$	Policy for operators
5.	$\langle go \rangle$	1, 2, 3 Reps
6.	$\langle launch \rangle$	5, 4 Modus Ponens
7.	<i>Bob</i>   <i>Operator</i> says $\langle launch \rangle$	6 Says

**Figure 14.4** Proof Justifying Application's Action

1.	<i>Operator</i> controls $\langle launch \rangle$	Operator's Authority
2.	<i>Bob</i> reps <i>Operator</i> on $\langle launch \rangle$	Bob is recognized as the Operator
3.	<i>Bob</i>   <i>Operator</i> says $\langle launch \rangle$	Bob's order
4.	$\langle launch \rangle \supset \langle activate \rangle$	Policy for applications
5.	$\langle launch \rangle$	1, 2, 3 Reps
6.	$\langle activate \rangle$	5, 4 Modus Ponens

**Example 14.2**

The derived inference rule justifying the Application's decision to activate when receiving Operator Bob's *launch* command is

$$ApRule\ 1 \quad \frac{\begin{array}{c} Operator\ controls\ \langle launch \rangle \quad Bob\ reps\ Operator\ on\ \langle launch \rangle \\ Bob\ |\ Operator\ says\ \langle launch \rangle \quad \langle launch \rangle \supset \langle activate \rangle \end{array}}{\langle activate \rangle}$$

The proof of *ApRule 1* is shown in Figure 14.4. ◇

## 14.3 Verifying the Command and Control Example in HOL

In this section, we define and verify in HOL the simple command and control example of Section 14.2. In the next series of examples we define the theory *simpleConopsExampleTheory*, define the datatypes for commands, people, roles, and principals. We then prove HOL theorems corresponding to *OpRule 1* and *ApRule 1*.

**Example 14.3**

In this example, we define the types of *simpleConopsExampleTheory*. Do the following.

1. Create, as normal, the file *simpleConopsExampleScript.sml* that builds *simpleConopsExampleTheory*.
2. **Important:** In the file *Holmakefile*, assign to the variable `INCLUDES` the path to the ACL subdirectory containing your HOL-compiled access-control logic theory files.
3. Within *simpleConopsExampleScript.sml*, add the typical infrastructure, i.e., the `structure`, `new_theory`, and `export_theory` statements.

4. **Important:** in addition to the *HolKernel*, *boolLib*, *Parse*, and *bossLib* theories and structures required by *Holmake*, **open** the structures and theories *acl.infRules*, *aclrulesTheory*, and *aclDrulesTheory*.

5. Add to your script file the HOL commands defining the following datatypes.

```
commands = go | launch | activate
people = Alice | Bob
roles = Commander | Operator
principals = Staff people | Role roles | Ap num
```

6. In a terminal, execute *Holmake* in the subdirectory containing *simpleConopsExampleScript.sml* to make sure your script file compiles as expected.  $\diamond$

#### Example 14.4

Theorem [OpRule1\\_thm](#) below corresponds to the inference rule proved by “pencil and paper” in Figure 14.3.

[OpRule1\_thm]

```
⊢ (M, Oi, Os) sat Name (Role Commander) controls prop go ⇒
  (M, Oi, Os) sat
  reps (Name (Staff Alice)) (Name (Role Commander)) (prop go) ⇒
  (M, Oi, Os) sat
  Name (Staff Alice) quoting Name (Role Commander) says
  prop go ⇒
  (M, Oi, Os) sat prop go impf prop launch ⇒
  (M, Oi, Os) sat
  Name (Staff Bob) quoting Name (Role Operator) says
  prop launch
```

The code snippets below show three ways of proving the theorem.

1. A forward proof:

```
val OpRule1_thm =
let
  val th1 =
    ACLASSUM
    ‘‘((Name (Role Commander)) controls (prop go)):(commands,principals,'d,'e)Form‘‘
  val th2 =
    ACLASSUM
    ‘‘(reps (Name (Staff Alice))(Name(Role Commander)) (prop go))
    : (commands,principals,'d,'e)Form‘‘
  val th3 =
    ACLASSUM
    ‘‘((Name(Staff Alice)) quoting (Name(Role Commander)) says (prop go))
    : (commands,principals,'d,'e)Form‘‘
  val th4 =
    ACLASSUM ‘‘((prop go) impf (prop launch)) : (commands,principals,'d,'e)Form‘‘
  val th5 = REPS th2 th3 th1
  val th6 = ACLMP th5 th4
  val th7 =
    SAYS ‘‘((Name(Staff Bob)) quoting (Name(Role Operator)):(principals Princ‘‘ th6
  val th8 = DISCH(hd(hyp th4)) th7
```

```

val th9 = DISCH(hd(hyp th3)) th8
val th10 = DISCH(hd(hyp th2)) th9
in
  DISCH(hd(hyp th1)) th10
end;

```

## 2. A goal-oriented proof using PROVE\_TAC alone:

```

val OpRule1.thmA =
TAC_PROOF(
  ([], ``((M :(commands, 'b, principals, 'd, 'e) Kripke),(Oi :'d po),
    (Os :'e po)) sat
    Name (Role Commander) controls
    (prop go :(commands, principals, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    reps (Name (Staff Alice)) (Name (Role Commander))
    (prop go :(commands, principals, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    Name (Staff Alice) quoting Name (Role Commander) says
    (prop go :(commands, principals, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    (prop go :(commands, principals, 'd, 'e) Form) impf
    (prop launch :(commands, principals, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    Name (Staff Bob) quoting Name (Role Operator) says
    (prop launch :(commands, principals, 'd, 'e) Form)``),
  PROVE_TAC[Reps, Modus_Ponens, Says])

```

## 3. A goal-oriented proof using a combination of ACL tactics and forward proof using the forward inference rules on the terms in the assumption list.

```

val OpRule1.thmB =
TAC_PROOF(
  ([], ``((M :(commands, 'b, principals, 'd, 'e) Kripke),(Oi :'d po),
    (Os :'e po)) sat
    Name (Role Commander) controls
    (prop go :(commands, principals, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    reps (Name (Staff Alice)) (Name (Role Commander))
    (prop go :(commands, principals, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    Name (Staff Alice) quoting Name (Role Commander) says
    (prop go :(commands, principals, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    (prop go :(commands, principals, 'd, 'e) Form) impf
    (prop launch :(commands, principals, 'd, 'e) Form) ==>
    (M,Oi,Os) sat
    Name (Staff Bob) quoting Name (Role Operator) says
    (prop launch :(commands, principals, 'd, 'e) Form)``),
  REPEAT STRIP_TAC THEN
  ACL_SAYS_TAC THEN
  PAT_ASSUM
  `` (M,Oi,Os) sat
  reps (Name (Staff Alice)) (Name (Role Commander)) (prop go)``
  (fn th1 =>
    (PAT_ASSUM
      `` (M,Oi,Os) sat
      Name (Staff Alice) quoting Name (Role Commander) says prop go``
    (fn th2 =>
      PAT_ASSUM
      `` (M,Oi,Os) sat Name (Role Commander) controls prop go``
      (fn th3 => ASSUME_TAC(REPS th1 th2 th3)))))) THEN
  PAT_ASSUM

```

```

''(M,Oi,Os) sat prop go''
(fn th1 =>
  (PAT.ASSUM
    ''(M,Oi,Os) sat prop go impf prop launch''
    (fn th2 => PROVE.TAC[(ACLMP th1 th2)])))

```

The above proofs exemplify three general approaches to access-control logic proofs. ◇

### Example 14.5

Theorem [ApRule1\\_thm](#) below corresponds to the inference rules proved by “pencil and paper” in Figure 14.4.

[\[ApRule1\\_thm\]](#)

```

⊢ (M, Oi, Os) sat Name (Role Operator) controls prop launch ⇒
  (M, Oi, Os) sat
  reps (Name (Staff Bob)) (Name (Role Operator))
  (prop launch) ⇒
  (M, Oi, Os) sat
  Name (Staff Bob) quoting Name (Role Operator) says
  prop launch ⇒
  (M, Oi, Os) sat prop launch impf prop activate ⇒
  (M, Oi, Os) sat prop activate

```

The code snippets below show three ways of proving the theorem.

#### 1. A forward proof:

```

val ApRule1_thm =
let
  val th1 =
    ACLASSUM
    ''((Name (Role Operator)) controls (prop launch))
    : (commands, principals, 'd, 'e)Form''
  val th2 =
    ACLASSUM
    ''(reps (Name (Staff Bob)) (Name (Role Operator)) (prop launch))
    : (commands, principals, 'd, 'e)Form''
  val th3 =
    ACLASSUM
    ''((Name (Staff Bob)) quoting (Name (Role Operator)) says (prop launch))
    : (commands, principals, 'd, 'e)Form''
  val th4 = ACLASSUM
    ''((prop launch) impf (prop activate)) : (commands, principals, 'd, 'e)Form''
  val th5 = REPS th2 th3 th1
  val th6 = ACLMP th5 th4
  val th7 = DISCH(hd(hyp th4)) th6
  val th8 = DISCH(hd(hyp th3)) th7
  val th9 = DISCH(hd(hyp th2)) th8
in
  DISCH(hd(hyp th1)) th9
end;

```

#### 2. A goal-oriented proof using PROVE\_TAC alone:

```

val ApRule1.thmA =
TAC_PROOF(
  ([],
    ‘(M,Oi,Os) sat Name (Role Operator) controls prop launch ==>
      (M,Oi,Os) sat
      reps (Name (Staff Bob)) (Name (Role Operator)) (prop launch) ==>
      (M,Oi,Os) sat
      Name (Staff Bob) quoting Name (Role Operator) says prop launch ==>
      (M,Oi,Os) sat prop launch impf prop activate ==>
      (M,Oi,Os) sat prop activate ‘’,
    PROVE_TAC[Reps, Modus_Ponens])

```

3. A goal-oriented proof using a combination of ACL tactics and forward proof using the forward inference rules on the terms in the assumption list.

```

val ApRule1.thmB =
TAC_PROOF(
  ([],
    ‘((M :(commands, 'b, principals, 'd, 'e) Kripke),(Oi :'d po),
      (Os :'e po)) sat
      Name (Role Operator) controls
      (prop launch :(commands, principals, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      reps (Name (Staff Bob)) (Name (Role Operator))
      (prop launch :(commands, principals, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      Name (Staff Bob) quoting Name (Role Operator) says
      (prop launch :(commands, principals, 'd, 'e) Form) ==>
      (M,Oi,Os) sat
      (prop launch :(commands, principals, 'd, 'e) Form) impf
      (prop activate :(commands, principals, 'd, 'e) Form) ==>
      (M,Oi,Os) sat (prop activate :(commands, principals, 'd, 'e) Form)‘),
    REPEAT STRIP_TAC THEN
    PAT_ASSUM
    ‘(M,Oi,Os) sat
      reps (Name (Staff Bob)) (Name (Role Operator)) (prop launch)‘
    (fn th1 =>
      (PAT_ASSUM
        ‘(M,Oi,Os) sat
          Name (Staff Bob) quoting Name (Role Operator) says prop launch‘
        (fn th2 =>
          PAT_ASSUM
            ‘(M,Oi,Os) sat Name (Role Operator) controls prop launch‘
            (fn th3 => ASSUME_TAC(REPS th1 th2 th3)))))) THEN
    PAT_ASSUM
    ‘(M,Oi,Os) sat prop launch‘
    (fn th1 =>
      (PAT_ASSUM
        ‘(M,Oi,Os) sat prop launch impf prop activate‘
        (fn th2 => PROVE_TAC[(ACL_MP th1 th2)]))))

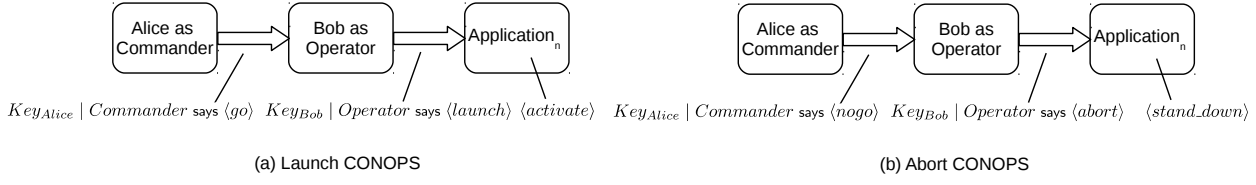
```

The above proofs exemplify three general approaches to access-control logic proofs. ◇

## 14.4 Exercises

### Exercise 14.4.1

Figure 14.5 shows a CONOPS similar to the one shown in Figure 14.2 with the following differences and additions.

**Figure 14.5** CONOPS Using Keys and Certificate Authorities

1. Commands issued by Commanders and Operators now use their respective cryptographic keys, in this case  $Key_{Alice}$  and  $Key_{Bob}$ .
2. Because cryptographic keys are used, a Certificate Authority CA issues key certificates for  $Key_{Alice}$  and  $Key_{Bob}$  signed by the CA's key  $Key_{CA}$ .
3. Besides the launch CONOPS, there is an additional abort CONOPS.
4. The abort CONOPS means that Commanders now can issue go and nogo commands; Operators can issue launch and abort commands; Applications can activate and stand\_down.

The above launch and abort CONOPS are characterized by the following four derived inference rules.

$$\begin{array}{l}
 \text{Commander controls } \langle go \rangle \quad \text{Alice reps Commander on } \langle go \rangle \\
 Key_{Alice} \mid \text{Commander says } \langle go \rangle \quad \langle go \rangle \supset \langle launch \rangle \\
 Key_{CA} \Rightarrow CA \quad Key_{CA} \text{ says } (Key_{Alice} \Rightarrow Alice) \\
 CA \text{ controls } (Key_{Alice} \Rightarrow Alice) \\
 \text{OpRuleLaunch} \quad \frac{}{Key_{Bob} \mid \text{Operator says } \langle launch \rangle}
 \end{array}$$

$$\begin{array}{l}
 \text{Operator controls } \langle launch \rangle \quad \text{Bob reps Operator on } \langle launch \rangle \\
 Key_{Bob} \mid \text{Operator says } \langle launch \rangle \quad \langle launch \rangle \supset \langle activate \rangle \\
 Key_{CA} \Rightarrow CA \quad Key_{CA} \text{ says } (Key_{Bob} \Rightarrow Bob) \\
 CA \text{ controls } (Key_{Bob} \Rightarrow Bob) \\
 \text{ApRuleActive} \quad \frac{}{\langle activate \rangle}
 \end{array}$$

$$\begin{array}{l}
 \text{Commander controls } \langle nogo \rangle \quad \text{Alice reps Commander on } \langle nogo \rangle \\
 Key_{Alice} \mid \text{Commander says } \langle nogo \rangle \quad \langle nogo \rangle \supset \langle abort \rangle \\
 Key_{CA} \Rightarrow CA \quad Key_{CA} \text{ says } (Key_{Alice} \Rightarrow Alice) \\
 CA \text{ controls } (Key_{Alice} \Rightarrow Alice) \\
 \text{OpRuleAbort} \quad \frac{}{Key_{Bob} \mid \text{Operator says } \langle abort \rangle}
 \end{array}$$

$$\begin{array}{l}
 \text{Operator controls } \langle abort \rangle \quad \text{Bob reps Operator on } \langle abort \rangle \\
 Key_{Bob} \mid \text{Operator says } \langle abort \rangle \quad \langle abort \rangle \supset \langle stand\_down \rangle \\
 Key_{CA} \Rightarrow CA \quad Key_{CA} \text{ says } (Key_{Bob} \Rightarrow Bob) \\
 CA \text{ controls } (Key_{Bob} \Rightarrow Bob) \\
 \text{ApRuleStandDown} \quad \frac{}{\langle stand\_down \rangle}
 \end{array}$$

**Your task is as follows.** Define a theory `conops0SolutionTheory`, which implements in HOL the CONOPS shown in Figure 14.5 and characterized by the four derived inference rules above.

Do the following.

1. Create `conops0SolutionTheory` using the following types, defined in the order shown below:

```

commands = go | nogo | launch | abort | activate | stand_down
people = Alice | Bob
roles = Commander | Operator | CA
keyPrinc =
  Staff conops0Solution$people
  | Role conops0Solution$roles
  | Ap num
principals = PR keyPrinc | Key keyPrinc

```

2. Prove in HOL theorems corresponding to the four derived inference rules. Use the following theorem names:

- (a) `OpRuleLaunch_thm`,
- (b) `OpRuleAbort_thm`,
- (c) `ApRuleActivate_thm`, and
- (d) `ApRuleStandDown_thm`

3. As a hint, we show the HOL theorem corresponding to `OpRuleLaunch_thm`. The other three theorems are similar.

`[OpRuleLaunch_thm]`

```

⊢ (M, Oi, Os) sat Name (PR (Role Commander)) controls prop go ⇒
  (M, Oi, Os) sat
  reps (Name (PR (Staff Alice))) (Name (PR (Role Commander)))
  (prop go) ⇒
  (M, Oi, Os) sat
  Name (Key (Staff Alice)) quoting
  Name (PR (Role Commander)) says prop go ⇒
  (M, Oi, Os) sat prop go impf prop launch ⇒
  (M, Oi, Os) sat
  Name (Key (Role CA)) speaks_for Name (PR (Role CA)) ⇒
  (M, Oi, Os) sat
  Name (Key (Role CA)) says
  Name (Key (Staff Alice)) speaks_for Name (PR (Staff Alice)) ⇒
  (M, Oi, Os) sat
  Name (PR (Role CA)) controls
  Name (Key (Staff Alice)) speaks_for Name (PR (Staff Alice)) ⇒
  (M, Oi, Os) sat
  Name (Key (Staff Bob)) quoting Name (PR (Role Operator)) says
  prop launch

```



## **Part V**

# **Lab Exercises: Cryptographic Components**



# Cryptographic Operations in HOL

---

At the core of many authentication, authorization, and confidentiality claims is the statement, “*we use cryptographic operations to assure integrity and confidentiality.*” This statement may satisfy people whose primary responsibilities do not include the technical details of how systems and system components are constructed. For engineers and computer scientists, whose responsibilities include design and verification, this statement is not good enough. Rigorous and reproducible assurance of authentication, authorization, and confidentiality require formal models, definitions, and proofs.

## 15.1 Properties, Reality, Purposes, and Models

We introduce the basic cryptographic operations of encryption and hashing in much the way computer hardware engineers view logic gates as components, software engineers view system calls and subroutines as building blocks, or users view applications. In what follows, we model cryptographic components and prove properties of the models. All models are simplified, incomplete, and inaccurate descriptions of reality. However, they are useful when they are “close enough” descriptions of reality and simplify the design and verification tasks of engineers and computer scientists.

**Properties** We view encryption and decryption algorithms as cryptographic components that hide or reveal information using cryptographic keys. *Symmetric*-key cryptographic components use the same key to hide and reveal information. *Asymmetric*-key cryptographic components use different keys to hide and reveal information. These are properties shared by all encryption methods. Encryption algorithms are deemed to be *strong* if the amount of effort required to reveal hidden information or deduce cryptographic keys is impractical.

Cryptographic hash functions take arbitrarily large inputs and produce fixed-sized outputs that are used to uniquely identify the input. A hash function is deemed to be *one way* if it is impractical to reverse, i.e., for any given hash value, it is impractical to determine an input that produces the given hash value.

**Reality** At the logic design and architecture levels of hardware design, details of signal delays and non-switch-level behavior are omitted. In reality, latches enter meta-stable states, where its outputs are neither zero nor one. Thus, gate level and register-transfer level descriptions are incomplete and imperfect models of reality. Nonetheless, their proven utility makes them essential design and verification tools for hardware design.

Cryptographic components are similar. Hash functions associate an infinite number of inputs with a finite number of hash values. When more than one input has the same hash value, this is called a *collision*. If collisions are impractical to predict and construct, we assume in practice that if two inputs have the same hash value, then the two inputs are identical. In reality, this is a crude approximation and different cryptographic algorithms have different strengths. Nevertheless, just as we ignore the details of meta-stability and time delays in digital hardware, we assume that the cryptographic strength of our components is sufficient for our purposes.

**Purposes** Cryptographic components are used for two purposes:

1. *Assurance of integrity*: information and commands are authenticated, i.e., their sources are known and their contents are free from corruption, and
2. *Assurance of confidentiality*: information and commands are accessible to only those who need to know.

How the above purposes are accomplished using the properties of cryptographic components is the subject of this laboratory. For example, we will show that one method to authenticate information is to combine a cryptographic hash with asymmetric encryption.

**Models** In what follows, we model cryptographic operations *algebraically*. We use symbols to represent cryptographic functions such as encryption functions, hash functions, keys, plain (unencrypted) text, and cipher (encrypted) text. The use of hash functions and encryption functions is described using algebraic data types. Accessing or decrypting the encrypted information using keys is defined by functions operating on datatypes corresponding to encrypted information.

The above approach leads to theorems describing the behavior of cryptographic operations based on the assumed properties of cryptographic components.

## 15.2 Building and Loading Cipher Theory

We build *cipherTheory* as follows. Do the following.

1. Put the files *cipherScript.sml*, *isainfRules.sml*, and *isainfRules.sig* in the same subdirectory.
2. In a terminal window, execute the following commands in the subdirectory containing the above files.

```
Holmake cleanAll
Holmake
```

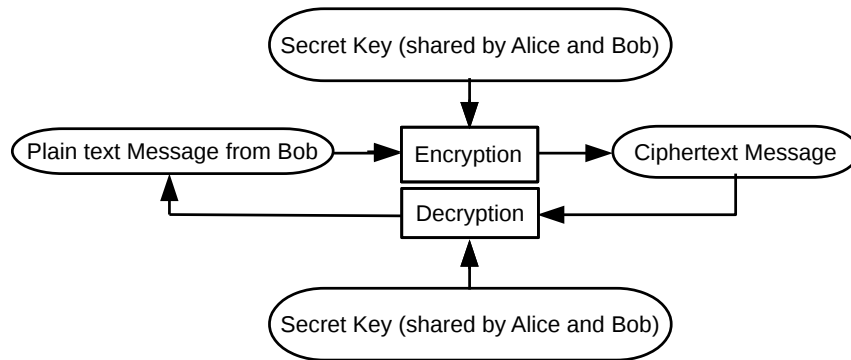
3. What you should see inside the terminal is shown below.

```
skchin@skVM:~/Desktop/Crypto$ Holmake cleanAll
skchin@skVM:~/Desktop/Crypto$ Holmake
/usr/local/share/HOL/bin/Holmake: Analysing cipherScript.sml
/usr/local/share/HOL/bin/Holmake: Trying to create directory .HOLMK for dependency files
/usr/local/share/HOL/bin/Holmake: Analysing isainfRules.sig
/usr/local/share/HOL/bin/Holmake: Analysing isainfRules.sml
Compiling isainfRules.sig
Compiling isainfRules.sml
Compiling cipherScript.sml

.... lots of output omitted ....

Exporting theory "cipher" ... done.
Theory "cipher" took 11.864s to build
/usr/local/share/HOL/bin/Holmake: Analysing cipherTheory.sml
/usr/local/share/HOL/bin/Holmake: Analysing cipherTheory.sig
Compiling cipherTheory.sig
Compiling cipherTheory.sml
skchin@skVM:~/Desktop/Crypto$
```

We access *cipherTheory* within HOL in the usual way. Do the following.

**Figure 15.1** Symmetric-Key Encryption and Decryption

1. Open a new or existing file in Emacs. Start a HOL session.
2. Within HOL, execute the following code.

```
load "cipherTheory";
open cipherTheory;
```

You should see the following.

```
- load "cipherTheory";
> val it = () : unit
- open cipherTheory;
> type thm = thm
  val asymMsg_nchotomy =
    |- !aa. ?p $o. aa = Ea p $o
    : thm

---- lots of output ----
-
```

2

In the sections that follow, we describe the contents of *cipherTheory*, which are models and properties of cryptographic operations, including symmetric encryption and decryption, asymmetric encryption and decryption, and digital signature generation and verification.

### 15.3 An Algebraic Model of Symmetric Key Encryption in HOL

Figure 15.1 is a schematic of symmetric key encryption and decryption. Suppose Bob wishes to send a message to Alice that only he and Alice can read. Also suppose that Bob and Alice share the same secret key, which is also known as a *symmetric key*. Here are the steps that Bob and Alice take to communicate confidentially.

1. Bob *encrypts* his message in plaintext with the secret key  $k$  he shares with Alice. He forwards to encrypted message, i.e., the ciphertext, to Alice.
2. Alice uses symmetric key  $k$  to decrypt the ciphertext to retrieve the plaintext message.

**Figure 15.2** Option Theory in HOL

---


$$\text{option} = \text{NONE} \mid \text{SOME } 'a$$


---

[option\_CLAUSES]

$$\begin{aligned} \vdash & (\forall x y. (\text{SOME } x = \text{SOME } y) \iff (x = y)) \wedge \\ & (\forall x. \text{THE } (\text{SOME } x) = x) \wedge (\forall x. \text{NONE} \neq \text{SOME } x) \wedge \\ & (\forall x. \text{SOME } x \neq \text{NONE}) \wedge (\forall x. \text{IS\_SOME } (\text{SOME } x) \iff \text{T}) \wedge \\ & (\text{IS\_SOME NONE} \iff \text{F}) \wedge (\forall x. \text{IS\_NONE } x \iff (x = \text{NONE})) \wedge \\ & (\forall x. \neg \text{IS\_SOME } x \iff (x = \text{NONE})) \wedge \\ & (\forall x. \text{IS\_SOME } x \Rightarrow (\text{SOME } (\text{THE } x) = x)) \wedge \\ & (\forall x. \text{option\_CASE } x \text{ NONE SOME} = x) \wedge \\ & (\forall x. \text{option\_CASE } x x \text{ SOME} = x) \wedge \\ & (\forall x. \text{IS\_NONE } x \Rightarrow (\text{option\_CASE } x e f = e)) \wedge \\ & (\forall x. \text{IS\_SOME } x \Rightarrow (\text{option\_CASE } x e f = f (\text{THE } x))) \wedge \\ & (\forall x. \text{IS\_SOME } x \Rightarrow (\text{option\_CASE } x e \text{ SOME} = x)) \wedge \\ & (\forall v f. \text{option\_CASE NONE } v f = v) \wedge \\ & (\forall x v f. \text{option\_CASE } (\text{SOME } x) v f = f x) \wedge \\ & (\forall f x. \text{OPTION\_MAP } f (\text{SOME } x) = \text{SOME } (f x)) \wedge \\ & (\forall f. \text{OPTION\_MAP } f \text{ NONE} = \text{NONE}) \wedge (\text{OPTION\_JOIN NONE} = \text{NONE}) \wedge \\ & \forall x. \text{OPTION\_JOIN } (\text{SOME } x) = x \end{aligned}$$


---

### 15.3.1 Idealized Behavior

Symmetric-key cryptography is used with the following expectations: (1) the same key is the only means to decrypt what is encrypted, (2) if something useful and recognizable is decrypted, then it must mean that the decrypted text and the decryption key are identical to the original text and encryption key, and (3) using anything other than the original encryption key to decrypt will result in an unusable result. We capture these expectations semi-formally by the following statements.

1. Whatever is encrypted with key  $k$  is retrieved unchanged by decrypting with the same key  $k$ .
2. If key  $k_1$  encrypted any plaintext, and key  $k_2$  decrypted the resulting ciphertext and retrieved the original text, then  $k_1 = k_2$ .
3. If plaintext is encrypted with key  $k_1$ , decrypted with key  $k_2$ , and nothing useful results, then  $k_1 \neq k_2$ .
4. If nothing useful is encrypted using any key, then nothing useful is decrypted using any key.

### 15.3.2 Modeling Idealized Behavior in HOL

**Figure 15.3** Definitions and Properties of Symmetric Encryption and Decryption

---


$$\text{symKey} = \text{sym num}$$

$$[\text{symKey\_one\_one}]$$

$$\vdash \forall a \ a'. \ (\text{sym } a = \text{sym } a') \iff (a = a')$$

$$\text{symMsg} = \text{Es symKey ('message option)}$$

$$[\text{symMsg\_one\_one}]$$

$$\vdash \forall a_0 \ a_1 \ a'_0 \ a'_1. \\ (\text{Es } a_0 \ a_1 = \text{Es } a'_0 \ a'_1) \iff (a_0 = a'_0) \wedge (a_1 = a'_1)$$

$$[\text{deciphS\_def}]$$

$$\vdash (\text{deciphS } k_1 \ (\text{Es } k_2 \ (\text{SOME } x)) = \\ \text{if } k_1 = k_2 \text{ then SOME } x \text{ else NONE}) \wedge \\ (\text{deciphS } k_1 \ (\text{Es } k_2 \ \text{NONE}) = \text{NONE})$$

$$[\text{deciphS\_clauses}]$$

$$\vdash (\forall k \ \text{text}. \text{deciphS } k \ (\text{Es } k \ (\text{SOME } \text{text})) = \text{SOME } \text{text}) \wedge \\ (\forall k_1 \ k_2 \ \text{text}. \\ (\text{deciphS } k_1 \ (\text{Es } k_2 \ (\text{SOME } \text{text})) = \text{SOME } \text{text}) \iff \\ (k_1 = k_2)) \wedge \\ (\forall k_1 \ k_2 \ \text{text}. \\ (\text{deciphS } k_1 \ (\text{Es } k_2 \ (\text{SOME } \text{text})) = \text{NONE}) \iff k_1 \neq k_2) \wedge \\ \forall k_1 \ k_2. \text{deciphS } k_1 \ (\text{Es } k_2 \ \text{NONE}) = \text{NONE})$$

$$[\text{deciphS\_one\_one}]$$

$$\vdash (\forall k_1 \ k_2 \ \text{text}_1 \ \text{text}_2. \\ (\text{deciphS } k_1 \ (\text{Es } k_2 \ (\text{SOME } \text{text}_2)) = \text{SOME } \text{text}_1) \iff \\ (k_1 = k_2) \wedge (\text{text}_1 = \text{text}_2)) \wedge \\ \forall \text{enMsg } \text{key} \ \text{enMsg}. \\ (\text{deciphS } \text{key} \ \text{enMsg} = \text{SOME } \text{text}) \iff \\ (\text{enMsg} = \text{Es } \text{key} \ (\text{SOME } \text{text}))$$


---

### Adding "Nothing Useful" as a Value

One aspect we must model is the notion of “nothing useful” as a value or result. To do this in a general fashion, we use *option* theory in HOL. Figure 15.2 shows the type definition of *option* and the properties of *option* types in HOL in the theorem *option\_CLAUSES*.

The *option* type is polymorphic. *option* types are created from other types using the type constructor *SOME*. For example, when *SOME* is applied to the natural number 1, i.e., *SOME 1*, the resulting value is of type *num option*. The *num option* type has all the values of *SOME n*, where *n* is a natural number in HOL, with one added value: *NONE*. We use *NONE* when we want to return a value other than a natural number, e.g., in the case where we return a result of dividing by zero.

In the case of modeling encryption and decryption, we use *option* types to add the value *NONE* to whatever we are encrypting or decrypting. Doing so allows us to handle cases such as what value to return if the wrong key is used to decrypt an encrypted message.

**Figure 15.4** Definition of Digests and their Properties

---

```
digest = hash ('message option)
```

```
[digest_one_one]
```

```
⊢ ∀a a'. (hash a = hash a') ⇔ (a = a')
```

---

Finally, the accessor function *THE* is used to retrieve the value to which *SOME* is applied. For example,  $THE(SOME\ x) = x$ , as shown in *option\_CLAUSES*.

### Symmetric Keys, Encryption, Decryption, and their Properties

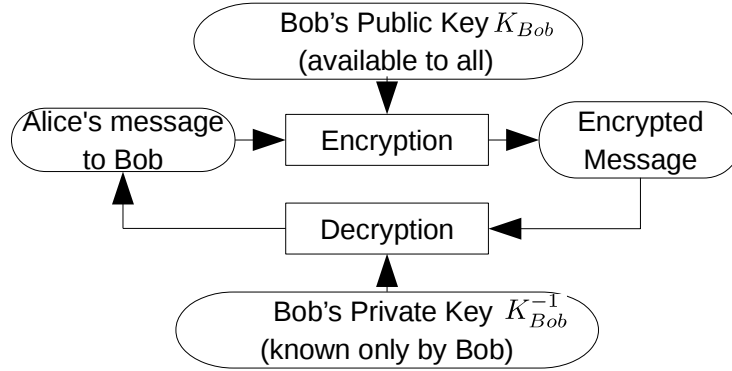
Figure 15.3 shows the definitions and properties of symmetric-key encryption and decryption. The following is a list of key definitions and properties.

- Symmetric keys are modeled by the algebraic type *symKey*. The type constructor is *sym*. For example, `sym 1234` is a symmetric key. Abstractly, `sym 1234` is the symmetric key which is identified by number 1234.
- Two symmetric keys are identical if they have the same number to which *sym* is applied. This is shown in theorem *symKey\_one\_one*.
- Symmetrically encrypted messages are modeled by the algebraic type *symMsg*, whose type constructor is *Es*. Symmetrically encrypted messages have two arguments: (1) a *symKey*, and (2) a 'message option'. For example, `Es (sym 1234) (SOME "This is a string")` is a symmetrically encrypted message using: (1) the symmetric key `sym 1234`, and (2) the *string option* value `SOME "This is a string"`. Abstractly, the type constructor *Es* stands for any symmetric-key encryption algorithm, e.g., DES or AES.
- Two *symMsg* values are identical if their corresponding components are identical. This is shown in theorem *symMsg\_one\_one*.
- Symmetric-key decryption of *symMsgs* is defined by *deciphS\_def*. If the same *symKey* is used to decipher an encrypted *SOME x*, then *SOME x* is returned. Otherwise, *NONE* is returned. If nothing useful is encrypted, then nothing useful is decrypted. Abstractly, *deciphS* represents any symmetric key decryption algorithm.
- Finally, *deciphS\_clauses* is the HOL theorem that shows our type definitions for keys and encryption, coupled with our definition of decryption, has the properties we expect: (1) the same key when used for encryption and decryption returns the original message, (2) if the original message was retrieved, identical keys were used, (3) if a different key is used to decrypt ciphertext, then nothing useful is returned, and (4) garbage in and garbage out holds true.

## 15.4 Cryptographic Hash Functions

Cryptographic hash functions are used to map inputs of any size into a fixed number of bits. Cryptographic hash functions are one-way functions, (1) the output is easy to compute from the input, and (2) it is computationally infeasible to determine an input when given only a hash value. Hash values are also known as *digests*.



**Figure 15.5** Asymmetric-Key Encryption and Decryption**Figure 15.6** Definitions and Properties of Asymmetric Keys and Messages

```
pKey = pubK 'princ | privK 'princ
```

```
[pKey_distinct_clauses]
```

$$\vdash (\forall a' a. \text{pubK } a \neq \text{privK } a') \wedge \forall a' a. \text{privK } a' \neq \text{pubK } a$$

```
[pKey_one_one]
```

$$\vdash (\forall a' a. (\text{pubK } a = \text{pubK } a') \iff (a = a')) \wedge$$

$$\forall a' a. (\text{privK } a = \text{privK } a') \iff (a = a')$$

```
asymMsg = Ea ('princ pKey) ('message option)
```

```
[asymMsg_one_one]
```

$$\vdash \forall a_0 a_1 a'_0 a'_1. \\ (\text{Ea } a_0 a_1 = \text{Ea } a'_0 a'_1) \iff (a_0 = a'_0) \wedge (a_1 = a'_1)$$

Figure 15.4 shows the type definition of *digest* and their properties. The following describes the type definition and its properties.

- Digests or hashes are modeled by the algebraic type *digest*. The type constructor is *hash* and is meant to represent any hash algorithm, e.g., SHA1 and SHA2. Notice that the *hash* is applied to polymorphic arguments of type *'message option*, e.g., `hash (SOME "A string message")`.
- The key property of *ideal* digests is they are one-to-one, as shown by the theorem *digest\_one\_one*. In reality, hashes cannot be one-to-one due to their fixed-length output. Modeling digests in this way is analogous to abstracting the electrical behavior of transistors as amplifiers away and idealizing them as perfect switches.

## 15.5 Asymmetric-Key Cryptography

Figure 15.5 is a schematic of asymmetric key encryption and decryption. The asymmetric nature of asymmetric key, or public-key cryptography, is two different keys are used instead of the same key. One

**Figure 15.7** Definitions and Properties of Asymmetric Decryption

[deciphP\_def]

$$\begin{aligned} \vdash & (\text{deciphP } \text{key} \ (\text{Ea} \ (\text{privK } P) \ (\text{SOME } x)) = \\ & \quad \text{if } \text{key} = \text{pubK } P \text{ then SOME } x \text{ else NONE}) \wedge \\ & (\text{deciphP } \text{key} \ (\text{Ea} \ (\text{pubK } P) \ (\text{SOME } x)) = \\ & \quad \text{if } \text{key} = \text{privK } P \text{ then SOME } x \text{ else NONE}) \wedge \\ & (\text{deciphP } k_1 \ (\text{Ea } k_2 \ \text{NONE}) = \text{NONE}) \end{aligned}$$

[deciphP\_clauses]

$$\begin{aligned} \vdash & (\forall P \ \text{text}. \\ & \quad (\text{deciphP} \ (\text{pubK } P) \ (\text{Ea} \ (\text{privK } P) \ (\text{SOME } \text{text})) = \\ & \quad \quad \text{SOME } \text{text}) \wedge \\ & \quad (\text{deciphP} \ (\text{privK } P) \ (\text{Ea} \ (\text{pubK } P) \ (\text{SOME } \text{text})) = \\ & \quad \quad \text{SOME } \text{text}) \wedge \\ & (\forall k \ P \ \text{text}. \\ & \quad (\text{deciphP } k \ (\text{Ea} \ (\text{privK } P) \ (\text{SOME } \text{text})) = \text{SOME } \text{text}) \iff \\ & \quad (k = \text{pubK } P)) \wedge \\ & (\forall k \ P \ \text{text}. \\ & \quad (\text{deciphP } k \ (\text{Ea} \ (\text{pubK } P) \ (\text{SOME } \text{text})) = \text{SOME } \text{text}) \iff \\ & \quad (k = \text{privK } P)) \wedge \\ & (\forall x \ k_2 \ k_1 \ P_2 \ P_1. \\ & \quad (\text{deciphP} \ (\text{pubK } P_1) \ (\text{Ea} \ (\text{pubK } P_2) \ (\text{SOME } x)) = \text{NONE}) \wedge \\ & \quad (\text{deciphP } k_1 \ (\text{Ea } k_2 \ \text{NONE}) = \text{NONE})) \wedge \\ & \forall x \ P_2 \ P_1. \text{deciphP} \ (\text{privK } P_1) \ (\text{Ea} \ (\text{privK } P_2) \ (\text{SOME } x)) = \text{NONE} \end{aligned}$$

key, known as a *public* key, may be freely disclosed. The other key, known as a *private* key, must be *known only by one principal*.

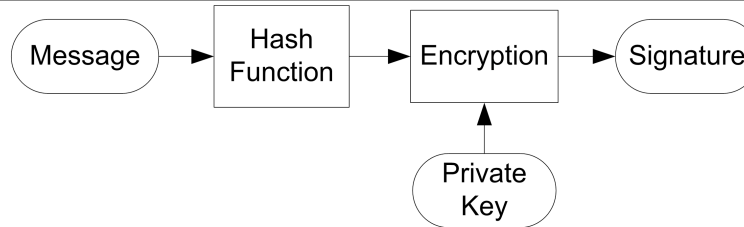
Suppose Alice wishes to send a message to Bob that only Bob can read. Alice encrypts the message to Bob using his public key  $K_{\text{Bob}}$ . Only Bob, who alone possesses the private key  $K_{\text{Bob}}^{-1}$ , is able to decrypt the message encrypted with his public key  $K_{\text{Bob}}$ .

Asymmetric-key cryptography is used with the following expectations: (1) plaintext that is encrypted with a private key and can be retrieved only with the corresponding public key, (2) plaintext that is encrypted with a public key can be retrieved only with the corresponding private key, (3) if plaintext was retrieved that was encrypted with a private key, then the corresponding public key was used to decrypt the ciphertext, (4) if plaintext was retrieved that was encrypted with a public key, then the corresponding private key was used to decrypt the ciphertext, and (5) nothing useful results if decryption uses anything but the corresponding public or private key used in encryption.

Figure 15.6 shows the type definitions for asymmetric keys *pKey*, i.e., public and private keys, and asymmetrically encrypted messages *asymMsg*. Figure 15.6 also shows properties of *pKey* and *asymMsg*.

- The type *pKey* has two forms, *pubK P* and *privK P*, public and private, respectively. Asymmetric keys are polymorphic and intended to be associated with principals *P* with variable type '*princ*'.
- The private and public keys of any principal are not the same.
- Public and private keys are the same if they have the same parameters.
- The type *asymMsg* represents asymmetrically encrypted messages. The parameters of type constructor *Ea* are a *pKey* and a '*message option*'. Abstractly, the type constructor *Ea* stands for any

**Figure 15.8** One-to-One Properties of Asymmetric Decryption[\[deciphP\\_one\\_one\]](#)

$$\begin{aligned}
&\vdash (\forall P_1 P_2 \text{ text}_1 \text{ text}_2. \\
&\quad (\text{deciphP } (\text{pubK } P_1) (\text{Ea } (\text{privK } P_2) (\text{SOME } \text{ text}_2)) = \\
&\quad \text{SOME } \text{ text}_1) \iff (P_1 = P_2) \wedge (\text{text}_1 = \text{text}_2) \wedge \\
&(\forall P_1 P_2 \text{ text}_1 \text{ text}_2. \\
&\quad (\text{deciphP } (\text{privK } P_1) (\text{Ea } (\text{pubK } P_2) (\text{SOME } \text{ text}_2)) = \\
&\quad \text{SOME } \text{ text}_1) \iff (P_1 = P_2) \wedge (\text{text}_1 = \text{text}_2) \wedge \\
&(\forall p \ c \ P \ \text{msg}. \\
&\quad (\text{deciphP } (\text{pubK } P) (\text{Ea } p \ c) = \text{SOME } \text{msg}) \iff \\
&\quad (p = \text{privK } P) \wedge (c = \text{SOME } \text{msg})) \wedge \\
&(\forall \text{enMsg } P \ \text{msg}. \\
&\quad (\text{deciphP } (\text{pubK } P) \ \text{enMsg} = \text{SOME } \text{msg}) \iff \\
&\quad (\text{enMsg} = \text{Ea } (\text{privK } P) (\text{SOME } \text{msg}))) \wedge \\
&(\forall p \ c \ P \ \text{msg}. \\
&\quad (\text{deciphP } (\text{privK } P) (\text{Ea } p \ c) = \text{SOME } \text{msg}) \iff \\
&\quad (p = \text{pubK } P) \wedge (c = \text{SOME } \text{msg})) \wedge \\
&\forall \text{enMsg } P \ \text{msg}. \\
&\quad (\text{deciphP } (\text{privK } P) \ \text{enMsg} = \text{SOME } \text{msg}) \iff \\
&\quad (\text{enMsg} = \text{Ea } (\text{pubK } P) (\text{SOME } \text{msg}))
\end{aligned}$$
**Figure 15.9** Digital Signature Generation

asymmetric-key algorithm, e.g., RSA.

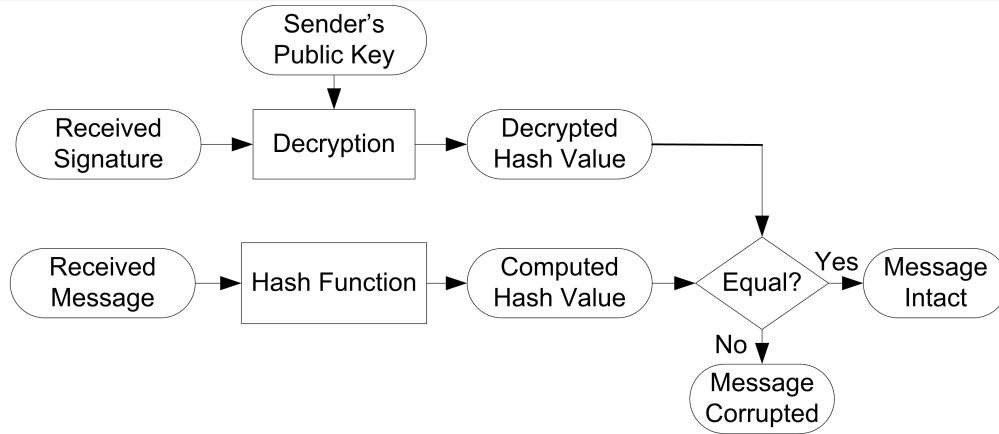
- Two *asymMsgs* are the same if they have the same *pKey* and *'message option* values.

Figure 15.7 shows the definition and properties of *deciphP*, which models the decryption of asymmetrically encrypted messages. Similar to symmetric-key encryption, to retrieve the plaintext *SOME x* requires use of the correct key, in this case *privK P* if the message was encrypted using *pubK P*, or *pubK P* if the message was encrypted with *privK P*. As before, garbage in produces garbage out.

The properties of *deciphP* are shown in Figures 15.7 and 15.8 by theorems *deciphP\_clauses* and *deciphP\_one\_one*. Together, they show the circumstances under which the original plaintext is decrypted, when nothing useful is decrypted, and the conditions that ensure that the expected keys and plaintext messages were in fact, used.

### 15.5.1 Digital Signatures

Digitally signed messages are often a combination of cryptographic hashes of messages encrypted using the *private key* of the sender. This is shown in Figure 15.9, which depicts signature generation as the following sequence of operations:

**Figure 15.10** Digital Signature Verification

1. A message is hashed, then
2. the message hash is encrypted using the private key of the sender.

The intuition behind signatures is this: (1) the cryptographic hash is a unique pointer to the message (and potentially much smaller than the message), and (2) encrypting using the sender's private key (which is reversible by the sender's public key) is a unique pointer to the sender.

Figure 15.10 shows how decrypted messages are checked for integrity using digital signatures. The top-most sequence from left to right shows how the decrypted hash value is retrieved from the received digital signature. The digital signature is decrypted using the sender's public key to retrieve the hash or digest of the original message. The retrieved hash is compared to the hash of the decrypted message. If the two hash values are the same, then the received message is judged to have arrived unchanged from the original.

Figure 15.11 shows the function definitions in HOL of *sign* and *signVerify*. *sign* takes as inputs a *pKey* and a digest and returns an asymmetrically encrypted digest using the asymmetric *pKey*. *signVerify* takes as input a *pKey*, digital signature, and a received message and compares the decrypted hash in the signature with the hash of the received message. The properties of *signVerify* and *sign* are in theorems *signVerifyOK* and *signVerify\_one\_one*.

- *signVerify* is always true for signatures generated as shown in Figure 15.9.
- *signVerify* and *sign* combine to have the desired properties that the plaintext must match and the corresponding keys must match.

**Figure 15.11** Digital Signature Generation, Verification, and Their Properties[\[sign\\_def\]](#)

$$\vdash \forall \text{pubKey } \text{dgst}. \text{ sign pubKey dgst} = \text{Ea pubKey (SOME dgst)}$$
[\[signVerify\\_def\]](#)

$$\vdash \forall \text{pubKey } \text{signature } \text{msgContents}. \\ \text{signVerify pubKey signature msgContents} \iff \\ (\text{SOME (hash msgContents)} = \text{deciphP pubKey signature})$$
[\[signVerifyOK\]](#)

$$\vdash \forall P \text{ msg}. \\ \text{signVerify (pubK P) (sign (privK P) (hash (SOME msg)))} \\ (\text{SOME msg})$$
[\[signVerify\\_one\\_one\]](#)

$$\vdash (\forall P \text{ m}_1 \text{ m}_2. \\ \text{signVerify (pubK P) (Ea (privK P) (SOME (hash (SOME m_1))))} \\ (\text{SOME m}_2) \iff (\text{m}_1 = \text{m}_2)) \wedge \\ (\forall \text{signature } P \text{ text}. \\ \text{signVerify (pubK P) signature (SOME text)} \iff \\ (\text{signature} = \text{sign (privK P) (hash (SOME text))})) \wedge \\ \forall \text{text}_2 \text{ text}_1 \text{ P}_2 \text{ P}_1. \\ \text{signVerify (pubK P}_1) (\text{sign (privK P}_2) (\text{hash (SOME text}_2))) \\ (\text{SOME text}_1) \iff (\text{P}_1 = \text{P}_2) \wedge (\text{text}_1 = \text{text}_2))$$

## 15.6 Exercises

**Exercise 15.6.1** Use the properties of symmetric key encryption and decryption in cipherTheory to prove the following theorems.

A. [\[exercisel5\\_6\\_1a\\_thm\]](#)

$$\vdash \forall \text{key } \text{enMsg } \text{message}. \\ (\text{deciphS key enMsg} = \text{SOME message}) \iff \\ (\text{enMsg} = \text{Es key (SOME message)})$$
B. [\[exercisel5\\_6\\_1b\\_thm\]](#)

$$\vdash \forall \text{keyAlice } k \text{ text}. \\ (\text{deciphS keyAlice (Es k (SOME text))} = \\ \text{SOME "This is from Alice"}) \iff \\ (k = \text{keyAlice}) \wedge (\text{text} = \text{"This is from Alice"})$$

**Exercise 15.6.2** Use the properties of asymmetric key encryption and decryption to prove the following theorems.

A. [\[exercisel5\\_6\\_2a\\_thm\]](#)

$$\vdash \forall P \text{ message}. \\ (\text{deciphP (pubK P) enMsg} = \text{SOME message}) \iff \\ (\text{enMsg} = \text{Ea (privK P) (SOME message)})$$

B. [\[exercisel5\\_6\\_2b\\_thm\]](#)

$$\begin{aligned} &\vdash \forall \text{key } \text{text} . \\ &\quad (\text{deciphP } (\text{pubK Alice}) \text{ (Ea key (SOME text))} = \\ &\quad \text{SOME "This is from Alice"}) \iff \\ &\quad (\text{key} = \text{privK Alice}) \wedge (\text{text} = \text{"This is from Alice"}) \end{aligned}$$

**Exercise 15.6.3** *Use the properties of signature verification to prove the following theorem.*

[\[exercisel5\\_6\\_3\\_thm\]](#)

$$\begin{aligned} &\vdash \forall \text{signature} . \\ &\quad \text{signVerify } (\text{pubK Alice}) \text{ signature} \\ &\quad (\text{SOME "This is from Alice"}) \iff \\ &\quad (\text{signature} = \\ &\quad \text{sign } (\text{privK Alice}) \text{ (hash (SOME "This is from Alice"))}) \end{aligned}$$

## **Part VI**

# **Lab Exercises: Transition Systems**





# High-Level State Machines

---

State machines are at the foundation of specifying, describing, and implementing systems with sequential behavior. The term *state* is intended to capture the notion that something can exist in different forms, modes, or configurations. For example, water has several familiar forms: liquid, solid, and gaseous. Each of these forms is viewed as a *state* in which water can be.

State machines allow us to satisfy the oft-needed requirement that a system give different responses to the same input. For example, some power tools have a simple mechanical “safe mode” on the trigger, a simple slider switch, which prevents the trigger from being squeezed. In the safe mode or state, the slider switch disables the tool when the trigger is squeezed. Moving the switch to the “operating mode” allows the trigger to activate the tool.

In this chapter we do the following:

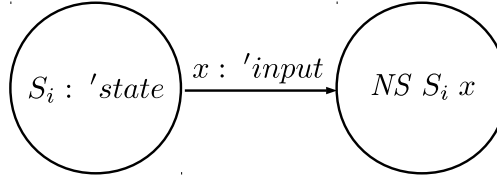
1. show how inductive relations are used to describe a state-transition system,
2. review the classical description of state machines in terms of their input alphabet, output alphabet, states, next-state transition function, and output function,
3. show how state machines are described with *labeled transition relations* and *configurations*,
4. devise a general parameterized framework, defined in HOL as *smTheory* that applies to all state machines (finite or otherwise), and
5. illustrate how *smTheory* is specialized to specific machines.

## 16.1 State Machines

State machines are widely used as controllers in digital hardware design. State machines control datapaths in central processing units (CPUs), they are at the core of the specialized controllers in car engines, brakes, and doors. They are deeply embedded in the control of industrial processes in manufacturing and critical infrastructure, such as petroleum refineries and electrical power grids. State machines in these applications provide special-purpose command and control of processes. These systems are known as SCADA (supervisory control and data acquisition) systems.

State machines are characterized by five components.

1. A set of states  $S = \{s_0, \dots, s_{n-1}, \dots\}$ ,
2. A set of inputs  $I = \{i_0, i_1, \dots, i_k, \dots\}$ ,
3. A set of outputs  $O = \{o_0, \dots, o_j, \dots\}$ ,
4. A next-state transition function  $\delta : S \rightarrow I \rightarrow S$ , and

**Figure 16.1** Parameterized State-Transition Relation

5. An output function  $\lambda : S \rightarrow I \rightarrow O$ , which is used to compute the **output in the next clock period**. Note: this is slightly different than the normal Mealy and Moore machine descriptions where  $\lambda$  computes the *current output*. We do this because of our use of input and output streams later on.

Suppose we wish to define state machines parametrically in terms of their state, input, and next-state transition functions, as shown in Figure 16.1. States and inputs are envisioned to be any type and each may have an infinite number of elements. We capture these notions by envisioning states  $S_i$  and inputs  $x$  to be polymorphic types in HOL, in this case represented by type variables  $: 'state$  and  $: 'input$ .

Returning to formalizing what is expressed graphically in Figure 16.1 by  $\xrightarrow{x}$ , we define a labeled transition relation *Trans*  $x$ , in words as follows.

1. For all next-state functions  $NS$ , inputs  $x$ , and states  $s$ , the predicate *Trans*  $x$  is true for states  $s$  and  $NS\ s\ x$ .
2. The set defining *Trans*  $x$  is the smallest set satisfying rule (1).

### 16.1.1 Defining Parameterized State Machines in HOL

We formally define what is described in words above and what is described graphically in Figure 16.1 in HOL. We do this by creating the HOL theory *smTheory*. Do the following.

#### Set up the script file *smScript.sml*

1. Create the HOL script file *smScript.sml*, which will build *smTheory*. Your theory will be based on the HOL structure *TypeBase* and the HOL theory *listTheory*. As usual, we will maintain *smTheory* using *Holmake*. This requires opening the usual HOL libraries and structures needed by *Holmake*: *HolKernel*, *boolLib*, *Parse*, and *bossLib*.
2. Your script file should look much like what follows.

```

structure smScript = struct

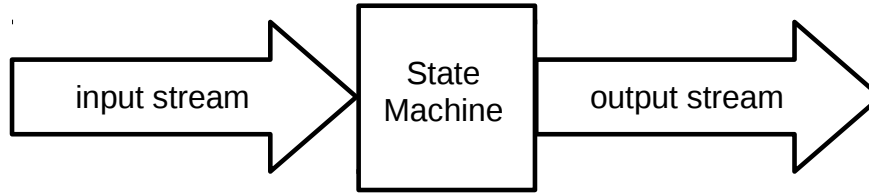
open HolKernel boolLib Parse bossLib
open TypeBase listTheory

val _ = new_theory "sm";

val _ = export_theory ();
val _ = print_theory "-";

end (* structure *)
  
```

3. As usual, run *Holmake* in a terminal in the same subdirectory as *smScript.sml* to make sure everything compiles correctly.

**Figure 16.2** Input and Output Streams

**Define inductively the labeled transition relation  $Trans\ x$**

1. Add the following code snippet to *smScript.sml* which defines the labeled transition relation  $Trans$  labeled with input  $x$ .

```

val (Trans_rules , Trans_ind , Trans_cases) =
  Hol_reln
  '!'NS (s:'state) (x:'input).  Trans x s ((NS:'state -> 'input -> 'state) s x) '

```

2. Executing the above definition produces the following three theorems.

```

[Trans_rules]
⊢ ∀NS s x. Trans x s (NS s x)

[Trans_ind]
⊢ ∀Trans'.
  (∀NS s x. Trans' x s (NS s x)) ⇒
  ∀a0 a1 a2. Trans a0 a1 a2 ⇒ Trans' a0 a1 a2

[Trans_cases]
⊢ ∀a0 a1 a2. Trans a0 a1 a2 ⇔ ∃NS. a2 = NS a1 a0

```

The theorem `Trans_rules` captures the conventional interpretation deterministic state machines and next-state functions as shown in Figure 16.1, i.e., states  $s$  and  $NS\ s\ x$  are related by the relation  $Trans\ x$ , for all values of  $s$  and  $x$ , and all next-state functions  $NS$ . Similarly, `Trans_cases` says that a labeled transition  $Trans\ a_0\ a_1\ a_2$  occurs if and only if the next state  $a_2$  is determined by some next-state function  $NS$  applied to state  $a_1$  and input  $a_0$ , and the transition label is the input  $a_0$ .

Notice that our definition is parameterized on all next-state functions  $NS$ . Our definition can describe all deterministic state machines. This is a benefit of using higher-order logic, which supports quantification over functions and predicates.

### 16.1.2 Defining State Machines Using Configurations

Another way of describing state machines is to view their behavior within the context of streams (lists) of inputs and outputs, as shown in Figure 16.2. We augment state-machine descriptions by including input and output streams (input lists and output lists) along with the current state of the machine.

In addition to the state-transition function  $\delta$  and the next output function  $\lambda$ , we define another labeled transition relation  $TR\ x$  (symbolized by  $\xrightarrow{x}$ ), where  $x$  is an input value, and  $\rightarrow$  is a transition relation on the *configuration* datatype. The algebraic type *configuration* has three components:

1. a list of inputs, the input stream,
2. a state, and

3. a list of outputs, the output stream.

Do the following by adding the following code to define the *configuration* datatype and inductively defined labeled transition relation *TR x* in HOL.

### Define the configuration datatype and prove its properties

1. Add the following code to *smScript.sml*:

```
(***)
(* Define configurations *)
(***)
val _ =
  Datatype
  'configuration = CFG ('input list) 'state ('output list)'

(***)
(* Note: configuration_11, configuration_induction, and configuration_nchotomy *)
(* are proved and available when Theory is loaded and opened *)
(***)
val configuration_11 = one_one_of ':(('input, 'state, 'output) configuration) '
val configuration_one_one = one_one_of ':(('input, 'state, 'output) configuration) '

val _ = save_thm("configuration_11", configuration_11)
val _ = save_thm("configuration_one_one", configuration_one_one)
```

Note: we have a duplicate naming of the same theorem—*configuration\_11* and *configuration\_one\_one*—solely for the purpose of having the one-to-one properties of *configuration* appear in its pretty-printed report as *configuration\_one\_one*.

2. Executing the above produces the new type and its associated properties shown below.

```
configuration = CFG ('input list) 'state ('output list)
[configuration_11] (and [configuration_one_one])
 $\vdash \forall a_0 \ a_1 \ a_2 \ a'_0 \ a'_1 \ a'_2. \\
\quad (CFG \ a_0 \ a_1 \ a_2 = CFG \ a'_0 \ a'_1 \ a'_2) \iff \\
\quad (a_0 = a'_0) \wedge (a_1 = a'_1) \wedge (a_2 = a'_2)$ 
```

3. As usual, execute *Holmake* in a terminal in the same subdirectory as *smScript.sml* to make sure everything compiles correctly.

### Define the Relation *TR x*

1. Add the following code to *smScript.sml*:

```
(***)
(* Define transition relation among configurations *)
(* This definition is parameterized in terms of *)
(* next state transition and output relations *)
(***)
val (TR_rules, TR_ind, TR_cases) =
  Hol_reln
  '!'NS Out (s:'state) (x:'input) (ins:'input list) (outs:'output list).
  TR x (CFG (x::ins) s outs)(CFG ins (NS s x) ((Out s x)::outs))'
```

2. Executing the above returns three theorems that characterize the transition relation *TR*.

```
[TR_rules]
```

$$\begin{aligned}
& \vdash \forall NS \text{ Out } s \ x \ ins \ outs. \\
& \quad TR \ x \ (CFG \ (x :: ins) \ s \ outs) \\
& \quad (CFG \ ins \ (NS \ s \ x) \ (Out \ s \ x :: outs)) \\
& [TR\_ind] \\
& \vdash \forall TR'. \\
& \quad (\forall NS \text{ Out } s \ x \ ins \ outs. \\
& \quad \quad TR' \ x \ (CFG \ (x :: ins) \ s \ outs) \\
& \quad \quad (CFG \ ins \ (NS \ s \ x) \ (Out \ s \ x :: outs))) \Rightarrow \\
& \quad \forall a_0 \ a_1 \ a_2. \ TR \ a_0 \ a_1 \ a_2 \Rightarrow TR' \ a_0 \ a_1 \ a_2 \\
& [TR\_cases] \\
& \vdash \forall a_0 \ a_1 \ a_2. \\
& \quad TR \ a_0 \ a_1 \ a_2 \iff \\
& \quad \exists NS \text{ Out } s \ ins \ outs. \\
& \quad \quad (a_1 = CFG \ (a_0 :: ins) \ s \ outs) \wedge \\
& \quad \quad (a_2 = CFG \ ins \ (NS \ s \ a_0) \ (Out \ s \ a_0 :: outs))
\end{aligned}$$

All three theorems are parameterized, where

- (a) *NS* is the next-state function,
- (b) *Out* is the next-output function,
- (c) *x* is the input at the head of the input stream  $x :: ins$ ,
- (d) *ins* is the tail of the input stream that comes after *x*, and
- (e) *outs* is the output stream.

The first theorem [TR\\_rules](#) relates a current configuration to its corresponding next configuration. The second theorem [TR\\_ind](#) is the induction principle based on *TR*. The third theorem states that  $TR \ a_0 \ a_1 \ a_2$  is true if and only if  $a_0$  is the head of the input stream  $a_0 :: ins$ , and there are next state and output functions that generate the next configuration.

3. As usual, within a terminal execute `Holmake` within the subdirectory containing `smScript.sml` to make sure everything compiles correctly.

### 16.1.3 Proving Equivalence Properties of *TR x*

The first property we prove characterizes  $TR \ x$  in terms of how the components of two configurations are related if the two configurations are related by  $TR \ x$ .

$$\begin{aligned}
& [TR\_clauses] \\
& \vdash (\forall x \ x1s \ s_1 \ out1s \ x2s \ out2s \ s_2. \\
& \quad TR \ x \ (CFG \ x1s \ s_1 \ out1s) \ (CFG \ x2s \ s_2 \ out2s) \iff \\
& \quad \exists NS \text{ Out } ins. \\
& \quad \quad (x1s = x :: ins) \wedge (x2s = ins) \wedge (s_2 = NS \ s_1 \ x) \wedge \\
& \quad \quad (out2s = Out \ s_1 \ x :: out1s)) \wedge \\
& \quad \forall NS \text{ Out } x \ x1s \ s_1 \ out1s \ x2s \ out2s. \\
& \quad \quad TR \ x \ (CFG \ x1s \ s_1 \ out1s) \\
& \quad \quad (CFG \ x2s \ (NS \ s_1 \ x) \ (Out \ s_1 \ x :: out2s)) \iff \\
& \quad \quad \exists ins. (x1s = x :: ins) \wedge (x2s = ins) \wedge (out2s = out1s)
\end{aligned}$$

Theorem *TR\_clauses* is proved by doing the following.

1. Add the following code to *smScript.sml*.

```
val lemma1 =
ISPECL [ 'x:'input'', 'CFG (x1s:'input list) (s1:'state) (out1s:'output list)'',
        'CFG (x2s:'input list) (s2:'state) (out2s:'output list)'' ] TR_cases
```

```
val lemma2 =
TAC.PROOF(
([],
  '!x x1s s1 out1s x2s out2s s2.
    TR (x:'input)
      (CFG (x1s:'input list) (s1:'state) (out1s:'output list))
      (CFG (x2s:'input list) (s2:'state) (out2s:'output list)) <=>
      ?NS Out ins.
        (x1s = x::ins) /\ (x2s = ins) /\ (s2 = NS s1 x) /\ (out2s = (Out s1 x)::out1s)'),
  REWRITE.TAC[lemma1, configuration_11, list_11] THEN
  REPEAT GEN.TAC THEN
  EQ.TAC THEN
  REPEAT STRIP.TAC THEN
  EXISTS.TAC 'NS:'state -> 'input -> 'state'' THEN
  EXISTS.TAC 'Out:'state -> 'input -> 'output'' THEN
  ASM.REWRITE.TAC[] THENL
  [(EXISTS.TAC 'ins:'input list'' THEN PROVE.TAC []),
   ALL.TAC] THEN
  EXISTS.TAC 's1:'state'' THEN
  EXISTS.TAC 'ins:'input list'' THEN
  EXISTS.TAC 'out1s:'output list'' THEN
  REWRITE.TAC[])
```

```
val lemma3 =
ISPECL [ 'x:'input'', 'CFG (x1s:'input list) (s1:'state) (out1s:'output list)'',
        'CFG
          (x2s:'input list)
          ((NS:'state -> 'input -> 'state) s1 x)
          ((Out:'state -> 'input -> 'output) s1 x::out2s)'' ] TR_cases
```

```
val lemma4 =
TAC.PROOF(([],
  '! (NS:'state -> 'input -> 'state) (Out:'state -> 'input -> 'output) (x:'input)
    (x1s:'input list) (s1:'state) out1s x2s out2s.
      TR (x:'input)
        (CFG (x1s:'input list) (s1:'state) (out1s:'output list))
        (CFG (x2s:'input list) (NS s1 x) (Out s1 x::out2s)) <=>
        ?ins. (x1s = x::ins) /\ (x2s = ins) /\ (out2s = out1s)'),
  REWRITE.TAC[lemma3, configuration_11, list_11] THEN
  REPEAT GEN.TAC THEN
  EQ.TAC THEN
  REPEAT STRIP.TAC THENL
  [(EXISTS.TAC 'ins:'input list'' THEN
    ASM.REWRITE.TAC[]),
   (EXISTS.TAC 'NS:'state -> 'input -> 'state'' THEN
    EXISTS.TAC 'Out:'state -> 'input -> 'output'' THEN
    EXISTS.TAC 's1:'state'' THEN
    EXISTS.TAC 'ins:'input list'' THEN
    EXISTS.TAC 'out1s:'output list'' THEN
    ASM.REWRITE.TAC[])]
```

```
val TR_clauses = CONJ lemma2 lemma4
val _ = save_thm("TR_clauses", TR_clauses)
```

2. As usual, we execute Holmake within a terminal in the subdirectory containing *smScript.sml* to make sure everything compiles correctly.

The next example provides a narrative of the proof of *TR\_clauses*.

### Example 16.1

In this example, we prove

[*TR\_clauses*]

$$\begin{aligned}
 &\vdash (\forall x \ x1s \ s_1 \ out1s \ x2s \ out2s \ s_2. \\
 &\quad TR \ x \ (CFG \ x1s \ s_1 \ out1s) \ (CFG \ x2s \ s_2 \ out2s) \iff \\
 &\quad \exists NS \ Out \ ins. \\
 &\quad (x1s = x :: ins) \wedge (x2s = ins) \wedge (s_2 = NS \ s_1 \ x) \wedge \\
 &\quad (out2s = Out \ s_1 \ x :: out1s) \wedge \\
 &\quad \forall NS \ Out \ x \ x1s \ s_1 \ out1s \ x2s \ out2s. \\
 &\quad TR \ x \ (CFG \ x1s \ s_1 \ out1s) \\
 &\quad (CFG \ x2s \ (NS \ s_1 \ x) \ (Out \ s_1 \ x :: out2s)) \iff \\
 &\quad \exists ins. (x1s = x :: ins) \wedge (x2s = ins) \wedge (out2s = out1s)
 \end{aligned}$$

What the theorem says is this:

1. An  $x$  transition from configuration `CFG x1s s1 out1s` to configuration `CFG x2s s2 out2s` occurs if and only if there exist the following conditions for some next-state function  $NS$ , some output function  $Out$ , and some list of inputs  $ins$ :
  - (a)  $x1s = x :: ins$ , i.e., that input  $x$  is the first element of input list  $x1s$ , and  $ins$  is the rest of the list  $x1s$  following  $x$ ,
  - (b)  $x2s = ins$ , i.e.,  $x2s$  is the rest of the list  $x1s$  after  $x$  is removed,
  - (c) state  $s_2 = NS \ s_1 \ x$ , i.e.  $s_2$  is determined by the next-state function  $NS$  applied to present state  $s_1$  and input  $x$ , and
  - (d) the output stream  $x2s = (Out \ s_1 \ x) :: out1s$ , i.e., the output based on the current state  $s_1$  and the current input  $x$  is pushed onto the list of previous outputs given by  $out1s$ .
2. An  $x$  transition from configuration `CFG x1s s1 out1s` to configuration `CFG x2s (NS s1 x) (Out s1 x :: out2s)` occurs if and only if there exist the following conditions for some input list  $ins$ 
  - (a)  $x1s = x :: ins$ , i.e., the first input of  $x1s$  is  $x$  and the remaining inputs are  $ins$ ,
  - (b)  $x2s = ins$ , i.e., after the  $x$  transition,  $x$  is removed from the input stream, and
  - (c) the output stream  $(Out \ s_1 \ x) :: out2s$  in the second configuration is just the output value  $Out \ s_1 \ x$  pushed onto the output stream  $out1s$  of the first configuration, i.e.,  $out2s = out1s$ .

The value of the above theorem is it gives the structure, values, and relationships among various forms and components of the first and second configurations related by transition relation  $TR \ x$ .

We develop the proof with four lemmas as follows. First, recall the theorem *TR\_cases*, which was automatically generated when we defined *TR*.

```

- TR_cases;
> val it =
  |- !a0 a1 a2.
    TR a0 a1 a2 <=>
      ?NS Out s ins outs.
      (a1 = CFG (a0::ins) s outs) /\
      (a2 = CFG ins (NS s a0) (Out s a0::outs))
: thm

```

The universally quantified variables `a0`, `a1` and `a2` are the input, starting configuration, and ending configuration. We specialize these variables to the values `'x:'input`, `CFG (x1s:'input list) (s1:'state) (out1s:'output list)`, and `CFG (x2s:'input list) (s2:'state) (out2s:'output list)`. Doing so gives us `lemma1`.

```
- val lemma1 =
ISPECL [``x:'input``, ``CFG (x1s:'input list) (s1:'state) (out1s:'output list)``,
        ``CFG (x2s:'input list) (s2:'state) (out2s:'output list)``] TR_cases;
> val lemma1 =
|- TR x (CFG x1s s1 out1s) (CFG x2s s2 out2s) <=>
   ?NS Out s ins outs.
   (CFG x1s s1 out1s = CFG (x::ins) s outs) /\
   (CFG x2s s2 out2s = CFG ins (NS s x) (Out s x::outs))
: thm
```

4

Given `lemma1`, we prove the first clause of `TR_clauses`. First, we set the goal.

```
- set_goal([],
``!x x1s s1 out1s x2s out2s s2.
  TR (x:'input)
    (CFG (x1s:'input list) (s1:'state) (out1s:'output list))
    (CFG (x2s:'input list) (s2:'state) (out2s:'output list)) <=>
  ?NS Out ins.
  (x1s = x::ins) /\ (x2s = ins) /\ (s2 = NS s1 x) /\
  (out2s = (Out s1 x)::out1s)``);
> val it =
Proof manager status: 1 proof.
1. Incomplete goalstack:
   Initial goal:

   !x x1s s1 out1s x2s out2s s2.
   TR x (CFG x1s s1 out1s) (CFG x2s s2 out2s) <=>
   ?NS Out ins.
   (x1s = x::ins) /\ (x2s = ins) /\ (s2 = NS s1 x) /\
   (out2s = Out s1 x::out1s)

: proofs
```

5

We rewrite the goal with `lemma1`.

```
- e(REWRITE_TAC[lemma1]);
OK..
1 subgoal:
> val it =

!x x1s s1 out1s x2s out2s s2.
(?NS Out s ins outs.
  (CFG x1s s1 out1s = CFG (x::ins) s outs) /\
  (CFG x2s s2 out2s = CFG ins (NS s x) (Out s x::outs))) <=>
?NS Out ins.
  (x1s = x::ins) /\ (x2s = ins) /\ (s2 = NS s1 x) /\
  (out2s = Out s1 x::out1s)

: proof
```

6

Given the above subgoal, we see an opportunity to take advantage of the one-to-one properties of configurations and lists. Recall the theorems `configuration_11` and `list_11`.



[configuration\_11]

$$\vdash \forall a_0 a_1 a_2 a'_0 a'_1 a'_2. (\text{CFG } a_0 a_1 a_2 = \text{CFG } a'_0 a'_1 a'_2) \iff (a_0 = a'_0) \wedge (a_1 = a'_1) \wedge (a_2 = a'_2)$$

[list\_11]

$$\vdash \forall a_0 a_1 a'_0 a'_1. (a_0 :: a_1 = a'_0 :: a'_1) \iff (a_0 = a'_0) \wedge (a_1 = a'_1)$$

We rewrite the subgoal using the two theorems to simplify the subgoal down to equalities among the components of configurations and lists.

```
- e(REWRITE_TAC[configuration_11, list_11]);
OK..
1 subgoal:
> val it =

!x x1s s1 out1s x2s out2s s2.
  (?NS Out s ins outs.
    ((x1s = x::ins) /\ (s1 = s) /\ (out1s = outs)) /\ (x2s = ins) /\
    (s2 = NS s x) /\ (out2s = Out s x::outs)) <=>
  ?NS Out ins.
    (x1s = x::ins) /\ (x2s = ins) /\ (s2 = NS s1 x) /\
    (out2s = Out s1 x::out1s)

: proof
```

7

Looking at the new subgoal, we eliminate all of the universally quantified variables by [REPEAT GEN\\_TAC](#), we divide the biconditional into its two implications using [EQ\\_TAC](#), we simplify both subgoals by [REPEAT STRIP\\_TAC](#), select *NS* and *Out* as our next state and output functions for both subgoals, and rewrite both subgoals using the assumptions. This is shown below.

```
- e(
  REPEAT GEN_TAC THEN
  EQ_TAC THEN
  REPEAT STRIP_TAC THEN
  EXISTS_TAC ``NS:'state -> 'input -> 'state`` THEN
  EXISTS_TAC ``Out:'state -> 'input -> 'output`` THEN
  ASM_REWRITE_TAC[]);
OK..
2 subgoals:
> val it =

?s ins' outs.
  ((x::ins = x::ins') /\ (s1 = s) /\ (out1s = outs)) /\ (ins = ins') /\
  (NS s1 x = NS s x) /\ (Out s1 x::out1s = Out s x::outs)
-----
0. x1s = x::ins
1. x2s = ins
2. s2 = NS s1 x
3. out2s = Out s1 x::out1s

?s ins'. (x::ins = x::ins') /\ (ins = ins')
-----
0. x1s = x::ins
1. s1 = s
2. out1s = outs
3. x2s = ins
4. s2 = NS s x
5. out2s = Out s x::outs

2 subgoals
: proof
```

8

The first subgoal is proved by selecting `ins:'input list` and then using `PROVE_TAC[]` to handle the rest of the details.

9

```

- e(
  EXISTS_TAC ``ins:'input list`` THEN
  PROVE_TAC []
);
OK..
Meson search level: ....

Goal proved.
[.....] |- ?ins'. (x::ins = x::ins') /\ (ins = ins')

Remaining subgoals:
> val it =

  ?s ins' outs.
  ((x::ins = x::ins') /\ (s1 = s) /\ (out1s = outs)) /\ (ins = ins') /\
  (NS s1 x = NS s x) /\ (Out s1 x::out1s = Out s x::outs)
  -----
  0. x1s = x::ins
  1. x2s = ins
  2. s2 = NS s1 x
  3. out

```

We deal with the remaining subgoal by choosing values for the existentially quantified variables that will yield the desired equalities within the subgoal, followed by rewriting to simplify and prove the resulting subgoal.

10

```

- e(
  EXISTS_TAC ``s1:'state`` THEN
  EXISTS_TAC ``ins:'input list`` THEN
  EXISTS_TAC ``out1s:'output list`` THEN
  REWRITE_TAC[]);
OK..

.... some output omitted ....

> val it =
  Initial goal proved.
  |- !x x1s s1 out1s x2s out2s s2.
    TR x (CFG x1s s1 out1s) (CFG x2s s2 out2s) <=>
    ?NS Out ins.
      (x1s = x::ins) /\ (x2s = ins) /\ (s2 = NS s1 x) /\
      (out2s = Out s1 x::out1s)
  : proof

```

As always, we bundle everything up into a single tactic to prove `lemma2`.

11

```

- val lemma2 =
TAC_PROOF(
  ([],
    ``!x x1s s1 out1s x2s out2s s2.
      TR (x:'input) (CFG (x1s:'input list) (s1:'state) (out1s:'output list))
        (CFG (x2s:'input list) (s2:'state) (out2s:'output list)) <=>
        ?NS Out ins.(x1s = x::ins) /\ (x2s = ins) /\ (s2 = NS s1 x) /\
        (out2s = (Out s1 x)::out1s)`),
    REWRITE_TAC[lemmal,configuration_11,list_11] THEN
    REPEAT GEN_TAC THEN
    EQ_TAC THEN
    REPEAT STRIP_TAC THEN
    EXISTS_TAC ``NS:'state -> 'input -> 'state`` THEN
    EXISTS_TAC ``Out:'state -> 'input -> 'output`` THEN
    ASM_REWRITE_TAC[] THENL
    [(EXISTS_TAC``ins:'input list`` THEN PROVE_TAC []),
     ALL_TAC] THEN
    EXISTS_TAC``s1:'state`` THEN
    EXISTS_TAC``ins:'input list`` THEN
    EXISTS_TAC``out1s:'output list`` THEN
    REWRITE_TAC[]);
Meson search level: ....
> val lemma2 =
  |- !x x1s s1 out1s x2s out2s s2.
    TR x (CFG x1s s1 out1s) (CFG x2s s2 out2s) <=>
    ?NS Out ins.
      (x1s = x::ins) /\ (x2s = ins) /\ (s2 = NS s1 x) /\
      (out2s = Out s1 x::out1s)
  : thm

```

With the proof of the first clause of [TR\\_clauses](#) done, we move on to prove the second clause.

The proof of the second clause is done in much the same way as the first clause. First, we specialize [TR\\_cases](#) to match the form of the transition relation term in the second clause. We do this as follows.

12

```

- val lemma3 =
ISPECL [``x:'input``, ``CFG (x1s:'input list) (s1:'state) (out1s:'output list)``,
  ``CFG
    (x2s:'input list)
    ((NS:'state -> 'input -> 'state) s1 x)
    ((Out:'state -> 'input -> 'output) s1 x::out2s)``] TR_cases;
> val lemma3 =
  |- TR x (CFG x1s s1 out1s) (CFG x2s (NS s1 x) (Out s1 x::out2s)) <=>
  ?NS' Out' s ins outs.
    (CFG x1s s1 out1s = CFG (x::ins) s outs) /\
    (CFG x2s (NS s1 x) (Out s1 x::out2s) =
     CFG ins (NS' s x) (Out' s x::outs))
  : thm

```

The proof of the second clause follows exactly the same strategy as the proof of the first clause: use [lemma3](#) to rewrite the initial goal followed by selecting values of terms that make the equalities work. The proof of the second clause ([lemma4](#)) is shown below.

13

```

- val lemma4 =
TAC_PROOF([[],
  ``!(NS:'state -> 'input -> 'state) (Out:'state -> 'input -> 'output) (x:'input)
    (x1s:'input list) (s1:'state) out1s x2s out2s.
    TR (x:'input) (CFG (x1s:'input list) (s1:'state) (out1s:'output list))
      (CFG (x2s:'input list) (NS s1 x) (Out s1 x::out2s)) <=>
      ?ins. (x1s = x::ins) /\ (x2s = ins) /\ (out2s = out1s)`),
REWRITE_TAC[lemma3, configuration_l1, list_l1] THEN
REPEAT GEN_TAC THEN
EQ_TAC THEN
REPEAT STRIP_TAC THENL
[(EXISTS_TAC ``ins:'input list`` THEN
  ASM_REWRITE_TAC[]),
 (EXISTS_TAC ``NS:'state -> 'input -> 'state`` THEN
  EXISTS_TAC ``Out:'state -> 'input -> 'output`` THEN
  EXISTS_TAC ``s1:'state`` THEN
  EXISTS_TAC ``ins:'input list`` THEN
  EXISTS_TAC ``out1s:'output list`` THEN
  ASM_REWRITE_TAC[])]);
> val lemma4 =
  |- !NS Out x x1s s1 out1s x2s out2s.
    TR x (CFG x1s s1 out1s) (CFG x2s (NS s1 x) (Out s1 x::out2s)) <=>
    ?ins. (x1s = x::ins) /\ (x2s = ins) /\ (out2s = out1s)
  : thm

```

We finish the proof by conjoining [lemma2](#) and [lemma4](#), as shown below.

14

```

- val TR_clauses = CONJ lemma2 lemma4;
> val TR_clauses =
  |- (!x x1s s1 out1s x2s out2s s2.
    TR x (CFG x1s s1 out1s) (CFG x2s s2 out2s) <=>
    ?NS Out ins.
      (x1s = x::ins) /\ (x2s = ins) /\ (s2 = NS s1 x) /\
      (out2s = Out s1 x::out1s)) /\
    !NS Out x x1s s1 out1s x2s out2s.
      TR x (CFG x1s s1 out1s) (CFG x2s (NS s1 x) (Out s1 x::out2s)) <=>
      ?ins. (x1s = x::ins) /\ (x2s = ins) /\ (out2s = out1s)
  : thm

```

At this point, we have proved [TR\\_clauses](#). The value of this theorem is that states the behavior of the transition relation  $TR\ x$  as it relates to specific configuration forms.  $\diamond$

### 16.1.4 Proving $TR\ x$ is Deterministic

Next, we prove a property that states  $TR\ x$  is deterministic, i.e., the next state and output components of configurations are determined by their next state functions  $NS$ , and output functions  $Out$ .

[\[TR\\_deterministic\]](#)

$$\begin{aligned}
 &\vdash \forall NS\ Out\ x_1\ ins_1\ s_1\ outs_1\ ins_2\ ins'_2\ outs_2\ outs'_2. \\
 &\quad TR\ x_1\ (CFG\ (x_1::ins_1)\ s_1\ outs_1) \\
 &\quad (CFG\ ins_2\ (NS\ s_1\ x_1)\ (Out\ s_1\ x_1::outs_2)) \wedge \\
 &\quad TR\ x_1\ (CFG\ (x_1::ins_1)\ s_1\ outs_1) \\
 &\quad (CFG\ ins'_2\ (NS\ s_1\ x_1)\ (Out\ s_1\ x_1::outs'_2)) \iff \\
 &\quad (CFG\ ins_2\ (NS\ s_1\ x_1)\ (Out\ s_1\ x_1::outs_2) = \\
 &\quad CFG\ ins'_2\ (NS\ s_1\ x_1)\ (Out\ s_1\ x_1::outs'_2)) \wedge \\
 &\quad TR\ x_1\ (CFG\ (x_1::ins_1)\ s_1\ outs_1) \\
 &\quad (CFG\ ins_2\ (NS\ s_1\ x_1)\ (Out\ s_1\ x_1::outs_2))
 \end{aligned}$$

The theorem states the following: if configurations `CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2)` and `CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2')` are both related by `TR x1` to configuration `CFG (x1::ins) (s1 outs1)`, then both configurations are identical, and vice versa (assuming `TR x1 (CFG (x1::ins) s1 outs1) (CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))`).

Theorem *TR\_deterministic* is proved by doing the following.

1. Add the following code to *smScript.sml*.

```
(*****
(* Proof that TR is deterministic *)
*****)
val lemma1 =
TAC.PROOF([],
  '!(NS:'state -> 'input -> 'state)(Out:'state -> 'input -> 'output).
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) ==>
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 '(NS s1 x1) ((Out s1 x1)::outs2'))) ==>
    (ins2 = ins2') /\ (outs2 = outs2')',
  REWRITE_TAC[TR_clauses] THEN
  REPEAT STRIP_TAC THEN
  ASM_REWRITE_TAC [] THEN
  IMP_RES_TAC list_11 THEN
  PROVE_TAC[])
```

```
val lemma2 =
TAC.PROOF([],
  '!(NS:'state -> 'input -> 'state)(Out:'state -> 'input -> 'output).
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) ==>
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 '(NS s1 x1) ((Out s1 x1)::outs2'))) ==>
    ((CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2)) =
     (CFG ins2 '(NS s1 x1) ((Out s1 x1)::outs2')))'',
  REWRITE_TAC[configuration_11, list_11] THEN
  REPEAT STRIP_TAC THEN
  IMP_RES_TAC lemma1)
```

```
val lemma3 =
TAC.PROOF([],
  '!(NS:'state -> 'input -> 'state)(Out:'state -> 'input -> 'output).
    ((CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2)) =
     (CFG ins2 '(NS s1 x1) ((Out s1 x1)::outs2')))' /\
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) ==>
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) /\
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 '(NS s1 x1) ((Out s1 x1)::outs2'))))' ',
  PROVE_TAC[])
```

```
val TR_deterministic =
TAC.PROOF([],
  '!(NS:'state -> 'input -> 'state)(Out:'state -> 'input -> 'output)
    x1 ins1 s1 outs1 ins2 ins2' outs2 outs2'.
    ((TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) /\
     (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 '(NS s1 x1) ((Out s1 x1)::outs2')))) =
    (((CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2)) =
     (CFG ins2 '(NS s1 x1) ((Out s1 x1)::outs2')))' /\
     (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))))' ',
  PROVE_TAC[lemma2, lemma3])

val _ = save_thm("TR_deterministic", TR_deterministic)
```

2. As usual, we execute *Holmake* within a terminal in the subdirectory containing *smScript.sml* to make sure everything compiles correctly.

The next example provides a narrative of the proof.

**Example 16.2**

The proof is done using three lemmas. The first lemma is as follows:

[lemma1]

$$\vdash \forall NS \text{ Out. } \text{TR } x_1 \text{ (CFG } (x_1::ins_1) \text{ s}_1 \text{ outs}_1) \text{ (CFG } ins_2 \text{ (NS } s_1 \text{ x}_1) \text{ (Out } s_1 \text{ x}_1::outs_2))} \Rightarrow \text{TR } x_1 \text{ (CFG } (x_1::ins_1) \text{ s}_1 \text{ outs}_1) \text{ (CFG } ins'_2 \text{ (NS } s_1 \text{ x}_1) \text{ (Out } s_1 \text{ x}_1::outs'_2))} \Rightarrow (ins_2 = ins'_2) \wedge (outs_2 = outs'_2)$$

Its proof is shown below.

```

- set_goal
([],
  ``!(NS:'state -> 'input -> 'state) (Out:'state -> 'input -> 'output).
    (TR x1 (CFG (x1::ins1) s1 outs1) (CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) ==>
    (TR x1 (CFG (x1::ins1) s1 outs1) (CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2'))) ==>
    (ins2 = ins2') /\ (outs2 = outs2')``);
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

    !NS Out.
      TR x1 (CFG (x1::ins1) s1 outs1)
        (CFG ins2 (NS s1 x1) (Out s1 x1::outs2)) ==>
      TR x1 (CFG (x1::ins1) s1 outs1)
        (CFG ins2' (NS s1 x1) (Out s1 x1::outs2')) ==>
      (ins2 = ins2') /\ (outs2 = outs2')

: proofs

```

15

Our first step is to rewrite the goal using [TR\\_clauses](#).

```

- e(REWRITE_TAC[TR_clauses]);
OK..
1 subgoal:
> val it =

  (?ins. (x1::ins1 = x1::ins) /\ (ins2 = ins) /\ (outs2 = outs1)) ==>
  (?ins. (x1::ins1 = x1::ins) /\ (ins2' = ins) /\ (outs2' = outs1)) ==>
  (ins2 = ins2') /\ (outs2 = outs2')

: proof

```

16

Next, we use [STRIP\\_TAC](#) repeatedly to simplify the goal, followed by rewriting with the assumptions.

```

- e(
  REPEAT STRIP_TAC THEN
  ASM_REWRITE_TAC []);
OK..
1 subgoal:
> val it =

  ins = ins'
  -----
  0. x1::ins1 = x1::ins
  1. ins2 = ins
  2. outs2 = outs1
  3. x1::ins1 = x1::ins'
  4. ins2' = ins'
  5. outs2' = outs1

: proof

```

17

We take advantage of the one-to-one properties of list as given by [list\\_11](#). We enrich the assumption list by resolving the assumptions with [list\\_11](#).

18

```
- e(IMP_RES_TAC list_11);
OK..
1 subgoal:
> val it =

  ins = ins'
  -----
  0. x1::ins1 = x1::ins
  1. ins2 = ins
  2. outs2 = outs1
  3. x1::ins1 = x1::ins'
  4. ins2' = ins'
  5. outs2' = outs1
  6. x1 = x1
  7. ins1 = ins'
  8. ins1 = ins
  9. !a1' a1. (a1 = a1') ==> (outs2'::a1 = outs1::a1')
  10. !a1' a1. (a1 = a1') ==> (ins2'::a1 = ins'::a1')
  11. !a1' a1. (a1 = a1') ==> ((x1::ins1)::a1 = (x1::ins')::a1')
  12. !a1' a1. (a1 = a1') ==> (outs2::a1 = outs1::a1')
  13. !a1' a1. (a1 = a1') ==> (ins2::a1 = ins::a1')
  14. !a1' a1. (a1 = a1') ==> ((x1::ins1)::a1 = (x1::ins)::a1')
  15. !a0' a0. (a0 = a0') ==> (a0::outs2' = a0'::outs1)
  16. x1::ins2' = x1::ins'
  17. x1::x1::ins1 = x1::x1::ins'
  18. !a0' a0. (a0 = a0') ==> (a0::outs2 = a0'::outs1)
  19. x1::ins2 = x1::ins
  20. x1::x1::ins1 = x1::x1::ins
: proof
```

We note that in assumptions 7 and 8 that  $ins1 = ins'$  and  $ins1 = ins$ . This is enough for [PROVE\\_TAC](#) to solve on its own.

19

```
- e(PROVE_TAC[]);
OK..
Meson search level: ....

.... output omitted ....

> val it =
  Initial goal proved.
|- !NS Out.
  TR x1 (CFG (x1::ins1) s1 outs1)
    (CFG ins2 (NS s1 x1) (Out s1 x1::outs2)) ==>
  TR x1 (CFG (x1::ins1) s1 outs1)
    (CFG ins2' (NS s1 x1) (Out s1 x1::outs2')) ==>
  (ins2 = ins2') /\ (outs2 = outs2')
: proof
```

As always we bundle up all the steps into one compound tactic.

20

```

- val lemma1 =
TAC_PROOF([[],
  ``!(NS:'state -> 'input -> 'state) (Out:'state -> 'input -> 'output).
    (TR x1 (CFG (x1::ins1) s1 outs1) (CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) ==>
    (TR x1 (CFG (x1::ins1) s1 outs1) (CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2')) ==>
    (ins2 = ins2') /\ (outs2 = outs2'))``,
  REWRITE_TAC[TR_clauses] THEN
  REPEAT STRIP_TAC THEN
  ASM_REWRITE_TAC [] THEN
  IMP_RES_TAC list_11 THEN
  PROVE_TAC[]);
Meson search level: ....
> val lemma1 =
  |- !NS Out.
    TR x1 (CFG (x1::ins1) s1 outs1)
      (CFG ins2 (NS s1 x1) (Out s1 x1::outs2)) ==>
    TR x1 (CFG (x1::ins1) s1 outs1)
      (CFG ins2' (NS s1 x1) (Out s1 x1::outs2')) ==>
    (ins2 = ins2') /\ (outs2 = outs2')
  : thm

```

Next, we prove [lemma2](#), which is closely related to [lemma1](#).

[[lemma2](#)]

$$\vdash \forall NS \text{ Out. } TR\ x_1\ (CFG\ (x_1::ins_1)\ s_1\ outs_1)\ (CFG\ ins_2\ (NS\ s_1\ x_1)\ (Out\ s_1\ x_1::outs_2)) \Rightarrow TR\ x_1\ (CFG\ (x_1::ins_1)\ s_1\ outs_1)\ (CFG\ ins'_2\ (NS\ s_1\ x_1)\ (Out\ s_1\ x_1::outs'_2)) \Rightarrow (CFG\ ins_2\ (NS\ s_1\ x_1)\ (Out\ s_1\ x_1::outs_2) = CFG\ ins'_2\ (NS\ s_1\ x_1)\ (Out\ s_1\ x_1::outs'_2))$$

First, we set the goal.

21

```

- set_goal
([],
  ``!(NS:'state -> 'input -> 'state) (Out:'state -> 'input -> 'output).
    (TR x1 (CFG (x1::ins1) s1 outs1) (CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) ==>
    (TR x1 (CFG (x1::ins1) s1 outs1) (CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2')) ==>
    ((CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2)) = (CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2'))))``);
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

    !NS Out.
    TR x1 (CFG (x1::ins1) s1 outs1)
      (CFG ins2 (NS s1 x1) (Out s1 x1::outs2)) ==>
    TR x1 (CFG (x1::ins1) s1 outs1)
      (CFG ins2' (NS s1 x1) (Out s1 x1::outs2')) ==>
    (CFG ins2 (NS s1 x1) (Out s1 x1::outs2) =
    CFG ins2' (NS s1 x1) (Out s1 x1::outs2'))

  : proofs

```

We simplify the goal by taking advantage of the one-to-one properties of configurations and lists.



22

```

- e(REWRITE_TAC[configuration_11,list_11]);
OK..
1 subgoal:
> val it =

!NS Out.
  TR x1 (CFG (x1::ins1) s1 outs1)
    (CFG ins2 (NS s1 x1) (Out s1 x1::outs2)) ==>
  TR x1 (CFG (x1::ins1) s1 outs1)
    (CFG ins2' (NS s1 x1) (Out s1 x1::outs2')) ==>
  (ins2 = ins2') /\ (outs2 = outs2')

: proof

```

Next, we use `STRIP_TAC` to simplify the subgoal and take advantage of `lemma1` using `IMP_RES_TAC`.

23

```

- e(
  REPEAT STRIP_TAC THEN
  IMP_RES_TAC lemma1);
OK..

... output omitted ...

> val it =
  Initial goal proved.
|- !NS Out.
  TR x1 (CFG (x1::ins1) s1 outs1)
    (CFG ins2 (NS s1 x1) (Out s1 x1::outs2)) ==>
  TR x1 (CFG (x1::ins1) s1 outs1)
    (CFG ins2' (NS s1 x1) (Out s1 x1::outs2')) ==>
  (CFG ins2 (NS s1 x1) (Out s1 x1::outs2) =
   CFG ins2' (NS s1 x1) (Out s1 x1::outs2'))

: proof

```

As always, we combine all the steps into a single tactic.

24

```

- val lemma2 =
  TAC_PROOF([[]],
    ``!(NS:'state -> 'input -> 'state) (Out:'state -> 'input -> 'output).
      (TR x1 (CFG (x1::ins1) s1 outs1) (CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) ==>
      (TR x1 (CFG (x1::ins1) s1 outs1) (CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2'))) ==>
      ((CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2)) = (CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2')) ``),
  REWRITE_TAC[configuration_11,list_11] THEN
  REPEAT STRIP_TAC THEN
  IMP_RES_TAC lemma1);
> val lemma2 =
  |- !NS Out.
    TR x1 (CFG (x1::ins1) s1 outs1)
      (CFG ins2 (NS s1 x1) (Out s1 x1::outs2)) ==>
    TR x1 (CFG (x1::ins1) s1 outs1)
      (CFG ins2' (NS s1 x1) (Out s1 x1::outs2')) ==>
    (CFG ins2 (NS s1 x1) (Out s1 x1::outs2) =
     CFG ins2' (NS s1 x1) (Out s1 x1::outs2'))

: thm

```

The third lemma is essentially the converse of `lemma2`.

[`lemma3`]

$$\vdash \forall NS \text{ Out. } (CFG \text{ ins}_2 (NS \text{ s}_1 \text{ x}_1) (Out \text{ s}_1 \text{ x}_1::outs_2) = CFG \text{ ins}'_2 (NS \text{ s}_1 \text{ x}_1) (Out \text{ s}_1 \text{ x}_1::outs'_2)) \wedge$$

$$TR \text{ x}_1 (CFG (x_1::ins_1) s_1 outs_1) (CFG \text{ ins}_2 (NS \text{ s}_1 \text{ x}_1) (Out \text{ s}_1 \text{ x}_1::outs_2)) \Rightarrow TR \text{ x}_1 (CFG (x_1::ins_1) s_1 outs_1) (CFG \text{ ins}_2 (NS \text{ s}_1 \text{ x}_1) (Out \text{ s}_1 \text{ x}_1::outs_2)) \wedge$$

$$TR \text{ x}_1 (CFG (x_1::ins_1) s_1 outs_1) (CFG \text{ ins}'_2 (NS \text{ s}_1 \text{ x}_1) (Out \text{ s}_1 \text{ x}_1::outs'_2))$$

This theorem is easily proved using `PROVE_TAC`.

25

```

- val lemma3 =
TAC_PROOF([[]],
  ``!(NS:'state -> 'input -> 'state)(Out:'state -> 'input -> 'output).
    ((CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2)) = (CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2'))) /\
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) ==>
    ((TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) /\
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2')))) ``),
PROVE_TAC[]);
Meson search level: .....
> val lemma3 =
  |- !NS Out.
    (CFG ins2 (NS s1 x1) (Out s1 x1::outs2) =
    CFG ins2' (NS s1 x1) (Out s1 x1::outs2')) /\
  TR x1 (CFG (x1::ins1) s1 outs1)
    (CFG ins2 (NS s1 x1) (Out s1 x1::outs2)) ==>
  TR x1 (CFG (x1::ins1) s1 outs1)
    (CFG ins2 (NS s1 x1) (Out s1 x1::outs2)) /\
  TR x1 (CFG (x1::ins1) s1 outs1)
    (CFG ins2' (NS s1 x1) (Out s1 x1::outs2'))
: thm

```

Finally, we prove the theorem `TR_deterministic` using `PROVE_TAC`, `lemma2` and `lemma3`.

26

```

val TR_deterministic =
TAC_PROOF([[]],
  ``!(NS:'state -> 'input -> 'state)(Out:'state -> 'input -> 'output)
    x1 ins1 s1 outs1 ins2 ins2' outs2 outs2'.
    ((TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2))) /\
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2')))) =
    (((CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2)) = (CFG ins2' (NS s1 x1) ((Out s1 x1)::outs2')) /\
    (TR x1 (CFG (x1::ins1) s1 outs1)(CFG ins2 (NS s1 x1) ((Out s1 x1)::outs2)))) ``),
PROVE_TAC[lemma2, lemma3])

```

At this point, we have proved `TR_deterministic`. The value of the theorem is that it states that for any configuration for which a `TR x` transition is possible, there is only one outcome.  $\diamond$

### 16.1.5 Proving `TR x` is Completely Specified

The theorem `TR_complete` states that `TR` is a completely specified relation. That is, for every configuration, there is a related configuration.

`[TR_complete]`

$$\vdash \forall s \ x \ ins \ outs. \\ \exists s' \ out. \\ TR \ x \ (CFG \ (x::ins) \ s \ outs) \ (CFG \ ins \ s' \ (out::outs))$$

What the above theorem states is that for all configurations `CFG (x::ins) s outs` there exists a state `s'` and output `out` such that `TR x (CFG (x::ins) s outs) (CFG ins s' (out::outs))`. The proof is as follows.

Theorem `TR_complete` is proved by doing the following.

1. Add the following code to `smScript.sml`.

```

val TR_complete =
TAC.PROOF([],
  ``!(s:'state)(x:'input)(ins:'input list)(outs:'output list).?(s':'state)(out:'output).
    (TR (x:'input)
      (CFG (x::ins) (s:'state) (outs:'output list))(CFG ins (s':'state) (out::outs)))``,
  REPEAT STRIP_TAC THEN
  REWRITE_TAC[TR_cases] THEN
  EXISTS_TAC``(NS:'state -> 'input -> 'state) s x`` THEN
  EXISTS_TAC``(Out:'state -> 'input -> 'output) s x`` THEN
  EXISTS_TAC``(NS:'state -> 'input -> 'state)`` THEN
  EXISTS_TAC``(Out:'state -> 'input -> 'output)`` THEN
  EXISTS_TAC``s:'state`` THEN
  EXISTS_TAC``ins:'input list`` THEN
  EXISTS_TAC``outs:'output list`` THEN
  REWRITE_TAC[])

val _ = save_thm("TR_complete",TR_complete)

```

2. As usual, we execute `Holmake` within a terminal in the subdirectory containing `smScript.sml` to make sure everything compiles correctly.

The following example provides a narrative of the proof.

### Example 16.3

First, we set the goal.

```

- set_goal
([],
  ``!(s:'state)(x:'input)(ins:'input list)(outs:'output list).?(s':'state)(out:'output).
    (TR (x:'input)
      (CFG (x::ins) (s:'state) (outs:'output list))
      (CFG ins (s':'state) (out::outs)))``);
> val it =
  Proof manager status: 1 proof.
  1. Incomplete goalstack:
    Initial goal:

    !s x ins outs.
      ?s' out. TR x (CFG (x::ins) s outs) (CFG ins s' (out::outs))

: proofs

```

27

We use `STRIP_TAC` to simplify the goal.

```

- e(REPEAT STRIP_TAC);
OK..
1 subgoal:
> val it =

  ?s' out. TR x (CFG (x::ins) s outs) (CFG ins s' (out::outs))

: proof

```

28

Next, we rewrite the goal using `TR_cases`.

```

- e(REWRITE_TAC[TR_cases]);
OK..
1 subgoal:
> val it =

  ?s' out NS Out s'' ins' outs'.
  (CFG (x::ins) s outs = CFG (x::ins') s'' outs') /\
  (CFG ins s' (out::outs) = CFG ins' (NS s'' x) (Out s'' x::outs'))

: proof

```

29

Next, we select a series of values to make the equalities match.

```

- e(
  EXISTS_TAC``(NS:'state -> 'input -> 'state) s x`` THEN
  EXISTS_TAC``(Out:'state -> 'input -> 'output) s x`` THEN
  EXISTS_TAC``(NS:'state -> 'input -> 'state)`` THEN
  EXISTS_TAC``(Out:'state -> 'input -> 'output)`` THEN
  EXISTS_TAC``s:'state`` THEN
  EXISTS_TAC``ins:'input list`` THEN
  EXISTS_TAC``outs:'output list`` );
OK..
1 subgoal:
> val it =

  (CFG (x::ins) s outs = CFG (x::ins) s outs) /\
  (CFG ins (NS s x) (Out s x::outs) = CFG ins (NS s x) (Out s x::outs))

: proof

```

30

As all the equalities are identical, we finish the proof using [REWRITE\\_TAC](#).

```

- e(REWRITE_TAC[]);
OK..

... output omitted ...

> val it =
  Initial goal proved.
|- !s x ins outs.
  ?s' out. TR x (CFG (x::ins) s outs) (CFG ins s' (out::outs))

: proof

```

31

As usual, we combine all of the above steps into a compound tactic.

32

```

- val TR_complete =
TAC_PROOF([[],
  ``!(s:'state) (x:'input) (ins:'input list) (outs:'output list).?(s' : 'state) (out:'output) .
    (TR (x:'input)
      (CFG (x::ins) (s:'state) (outs:'output list))
      (CFG ins (s' : 'state) (out::outs)))``),
REPEAT STRIP_TAC THEN
REWRITE_TAC[TR_cases] THEN
EXISTS_TAC``(NS:'state -> 'input -> 'state) s x`` THEN
EXISTS_TAC``(Out:'state -> 'input -> 'output) s x`` THEN
EXISTS_TAC``(NS:'state -> 'input -> 'state)`` THEN
EXISTS_TAC``(Out:'state -> 'input -> 'output)`` THEN
EXISTS_TAC``s:'state`` THEN
EXISTS_TAC``ins:'input list`` THEN
EXISTS_TAC``outs:'output list`` THEN
REWRITE_TAC[];
> val TR_complete =
  |- !s x ins outs.
    ?s' out. TR x (CFG (x::ins) s outs) (CFG ins s' (out::outs))
    : thm

```

At this point we have proved `TR_complete`. The value of the theorem is it establishes that all configurations have a related configuration.  $\diamond$

### 16.1.6 Proving the Equivalence of *Trans x* and *TR x*

The theorem *Trans\_Equiv\_TR* states that both relations are logically equivalent.

`[Trans_Equiv_TR]`

$$\vdash \text{TR } x \text{ (CFG (x::ins) s outs)} \\ (\text{CFG ins (NS s x) (Out s x::outs)}) \iff \text{Trans } x \text{ s (NS s x)}$$

The theorem is proved by doing the following.

1. Add the following code to *smScript.sml*.

```

(*****
(* Show trans and TR are equivalent *)
(*****)
val Trans_TR_lemma =
TAC.PROOF([[], `` (Trans (x:'input) (s:'state) (NS s x)) ==>
(TR x (CFG (x::ins) s (outs:'output list))(CFG ins (NS s x) ((Out s x)::outs))) ``),
STRIP_TAC THEN
PROVE_TAC[TR_rules])

val _ = save_thm("Trans_TR_lemma", Trans_TR_lemma)

```

```

val TR_Trans_lemma =
TAC.PROOF([[],
  `` (TR (x:'input)
    (CFG (x::ins) (s:'state) (outs:'output list))
    (CFG ins (NS s x) ((Out s x)::outs))) ==>
    (Trans (x:'input) (s:'state) (NS s x)) ``),
STRIP_TAC THEN
IMP_RES_TAC TR_cases THEN
PAT_ASSUM
  ``CFG (x::ins) s outs = CFG (x::ins') s' outs``
  (fn th => ASSUME_TAC(REWRITE_RULE[configuration_11, list_11] th)) THEN
PROVE_TAC[Trans_rules])

val _ = save_thm("TR_Trans_lemma", TR_Trans_lemma)

```

```

val Trans_Equiv_TR =
TAC_PROOF([[],
  ‘(TR (x:’input)
    (CFG (x::ins) (s:’state)(outs:’output list))
    (CFG ins (NS s x)((Out s x)::outs))) =
    (Trans (x:’input) (s:’state) (NS s x))’),
PROVE_TAC[TR_Trans_lemma,Trans_TR_lemma])

val _ = save_thm("Trans_Equiv_TR",Trans_Equiv_TR)

```

2. As usual, execute `Holmake` in a terminal in the subdirectory containing `smScript.sml` to make sure everything compiles correctly.

### 16.1.7 Defining Specialized Inference Rules for *smTheory*

As part of our state-machine infrastructure, we define three inference rules that instantiate specific next-state and output functions into three theorems of *smTheory*. We do this to provide support for users of *smTheory*.

We create two files: (1) *sminfRules.sml*, which defines the custom inference rules, and (2) *sminfRules.sig*, which specifies the type signatures for each inference rule. Both are needed by HOL to make the inference rules accessible to HOL when working with other theories.

**Defining *sminfRules.sml*** The inference rules based on the theorems of *smTheory* are defined in the file *sminfRules.sml*. Do the following.

1. Create the file *sminfRules.sml*.
2. Add the following code.

```

(*****
(* Inference rules for state machines *)
*****)
structure sminfRules :> sminfRules = struct

open HolKernel boolLib Parse bossLib
open reduceLib smTheory

fun SPEC_TR (ns:term) (out:term) = ISPECL [ns,out] TR_rules

fun SPEC_TR_clauses (ns:term) (out:term) =
CONJ
  (CONJUNCT1 TR_clauses)
  (ISPECL[ns,out](CONJUNCT2 TR_clauses))

fun SPEC_Trans_Equiv_TR (ns:term)(out:term) =
ISPECL
  [ns,out]
  (GENL
    [‘NS:’state -> ‘input -> ‘state ‘,
     ‘Out:’state -> ‘input -> ‘output ‘]
    Trans_Equiv_TR)

end; (* structure *)

```

The above file is very similar, but not identical, to the script files used to create HOL theories. The ML statement `structure sminfRules :> sminfRules = struct` at the beginning of *sminfRules.sml*, and the ML statement `end; (* structure *)` at the end of the file, are used by ML so that the function definitions in the body of the file are available for loading by HOL script files.

We maintain the inference rules using `Holmake`, so the theories required by `Holmake` are the ones used by script files, `open HolKernel boolLib Parse bossLib`. The inference rules we define are based on specializing some of the theorems in `smTheory`, so we need to make these available, which explains the code `open smTheory`. The inference rules themselves, `SPEC_TR`, `SPEC_TR_clauses`, and `SPEC_Trans_Equiv_TR`, are straightforward functions instantiating parameters `ns`—the next-state function, and `out`—the output function, into `smTheory` theorems `TR_rules`, `TR_clauses`, and `Trans_Equiv_TR`.

**Defining `sminfRules.sig`** In addition to the file `sminfRules.sml`, which defines the specialized inference rules based on `smTheory`, HOL needs the type signatures of each of the inference rules to compile the rules. The file `sminfRules.sig` supplies this information.

1. Create the file `sminfRules.sig`.
2. Add the following code.

```
signature sminfRules = sig
  type tactic = Abbrev.tactic;
  type thm_tactic = Abbrev.thm_tactic;
  type conv = Abbrev.conv;
  type thm = Thm.thm;
  type term = Term.term;
  type hol_type = Type.hol_type
  val SPEC_TR : term -> term -> thm
  val SPEC_TR_clauses : term -> term -> thm
  val SPEC_Trans_Equiv_TR : term -> term -> thm
end;
```

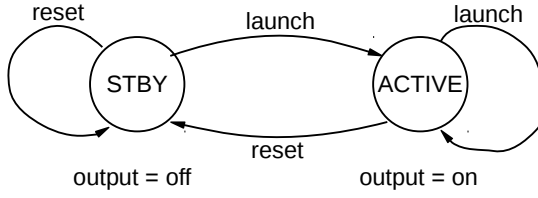
In the above code, the first and last lines inform ML that the file is a signature file. The **type** declarations declare the various HOL types used. The **val** statements declare the type signatures of our specialized inference rules `SPEC_TR`, `SPEC_TR_clauses`, and `SPEC_Trans_Equiv_TR`.

3. As usual, execute `Holmake` in a terminal in the subdirectory containing `sminfRules.sig` and `sminfRules.sml`, and make sure everything compiles with no errors.

## 16.2 Defining State Machines Using `smTheory`

When using `smTheory` to describe specific state machines, we do the following.

1. Define the datatypes for inputs, states, and outputs; prove theorems about the datatypes using the functions in `TypeBase`
2. Define the next-state transition function corresponding to `NS`; define the output function corresponding to `Out`
3. Specialize the definitions and theorems in `smTheory` with the above to describe the specific machine in which we are interested.

**Figure 16.3** Finite-State Machine Example  $M_0$ 

(a) State-Transition Diagram

Present State	Next State, Next Output	
	In = launch	In = reset
STBY	ACTIVE, on	STBY, off
ACTIVE	ACTIVE, on	STBY, off

*Machine  $M_0$* 

(b) State Table Description

### 16.2.1 Defining Machine $M_0$ Using *smTheory*

With *smTheory*, we have a parameterized infrastructure that enables us to describe conventional state machines. Consider Figure 16.3, which shows an example two-state machine  $M_0$  with the following definitions.

1.  $S = \{\text{STBY}, \text{ACTIVE}\}$
2.  $I = \{\text{launch}, \text{reset}\}$
3.  $O = \{\text{off}, \text{on}\}$
4. Next-state function  $\delta$  is defined as

$$\begin{aligned}
 \delta(\text{STBY}, \text{launch}) &= \text{ACTIVE} \\
 \delta(\text{STBY}, \text{reset}) &= \text{STBY} \\
 \delta(\text{ACTIVE}, \text{launch}) &= \text{ACTIVE} \\
 \delta(\text{ACTIVE}, \text{reset}) &= \text{STBY}
 \end{aligned}$$

5. Next-output function  $\lambda$  is defined as:

$$\begin{aligned}
 \lambda(\text{STBY}, \text{launch}) &= \text{on} \\
 \lambda(\text{STBY}, \text{reset}) &= \text{off} \\
 \lambda(\text{ACTIVE}, \text{launch}) &= \text{on} \\
 \lambda(\text{ACTIVE}, \text{reset}) &= \text{off}
 \end{aligned}$$

Figure 16.3(a) is a state-transition diagram describing  $M_0$ . The circles are states and the arcs represent state transitions associated with each input. Figure 16.3(b) is a tabular specification of  $M_0$  and contains the same information as the state-transition diagram. If  $M_0$  is in the *standby* (STBY) state, then the output of  $M_0$  is *off*. If the input is *reset*, then  $M_0$  remains in the STBY state. If the input is *launch*, then the next state is the *active* (ACTIVE) state. If  $M_0$  is in the ACTIVE state, then the output is *on*. If the input is *launch* then  $M_0$  remains in the ACTIVE state. If the input is *reset*, then the next state is STBY.

The following examples develop *m0Theory*, which formally describes  $M_0$  based on *smTheory*. Example 16.4 defines the datatypes for inputs, state, outputs, and their properties, which is the first step in our three-step process described above. Example 16.5 defines the next-state and output functions for machine  $M_0$ . Example 16.6 specializes the parameters of the general theorems in *smTheory* to the datatypes, next state, and output functions of  $M_0$ .



**Example 16.4**

In this example, we create the theory *m0Theory* and define the datatypes for inputs, states, and outputs, and their properties. Do the following.

1. As usual, create the file *m0Script.sml* and anticipate maintaining the theory using Holmake. The initial version of your file, before adding any definitions or proofs, should contain what is shown below. *m0Theory* depends on *TypeBase* and *smTheory*.

```
structure m0Script = struct

open HolKernel boolLib Parse bossLib
open TypeBase smTheory

val _ = new_theory "m0"

val _ = export_theory ()
val _ = print_theory "-"

end (* structure *)
```

2. Before adding anything else, run Holmake in a terminal in the same subdirectory in which *m0Script.sml* resides. Make sure, since *m0Theory* is built using *smTheory*, that your *Holmakefile* file has the path to *smTheory* assigned to INCLUDES.
3. Next, we define the datatypes for inputs, states, and outputs. We also prove the usual properties about them, namely that each element of each datatype is distinct from all the other members. Add the following code to *m0Script.sml*.

```
(* ----- *)
(* Define datatype for inputs                               *)
(* ----- *)
val _ =
  Datatype
  'command = launch | reset '

(* ----- *)
(* Prove distinctiveness properties of command using the function found in *)
(* in TypeBase                                             *)
(* ----- *)
val command_distinct_clauses = distinct_of '':command''
val _ = save_thm("command_distinct_clauses", command_distinct_clauses)
```

The above code introduces the following datatype and distinctiveness theorem.

*command* = launch | reset  
 [command\_distinct\_clauses]  
 $\vdash \text{launch} \neq \text{reset}$

```
(* ----- *)
(* Define the states                                       *)
(* ----- *)
val _ =
  Datatype 'state = STBY | ACTIVE'

(* ----- *)
(* Prove distinctiveness properties of state using the function found in *)
(* TypeBase                                               *)
(* ----- *)
val state_distinct_clauses = distinct_of '':state''
val _ = save_thm("state_distinct_clauses", state_distinct_clauses)
```

The above code introduces the following datatype and distinctiveness theorem.

```
state = STBY | ACTIVE
[state_distinct_clauses]
⊢ STBY ≠ ACTIVE
```

```
(* ----- *)
(* Define the outputs *)
(* ----- *)
val _ =
  Datatype 'output = on | off '

(* ----- *)
(* Prove distinctiveness properties of outputs using the function found in *)
(* TypeBase *)
(* ----- *)
val output_distinct_clauses = distinct_of ``:output``
val _ = save_thm("output_distinct_clauses", output_distinct_clauses)
```

The above code introduces the following datatype and distinctiveness theorem.

```
output = on | off
[output_distinct_clauses]
⊢ on ≠ off
```

4. Before adding anything else, run `Holmake` in a terminal in the same subdirectory in which `m0Script.sml` resides to make sure everything compiles correctly. ◇

### Example 16.5

In this example, we define the next state and output functions for machine  $M_0$ , as defined in Figure 16.3. This is the second step on specializing *smTheory* to describe and define machine  $M_0$ . Do the following.

1. Add the following code to `m0Script.sml`.

```
(* ----- *)
(* Define next-state function for machine M0 *)
(* ----- *)
val M0ns_def =
  Define '(M0ns STBY reset = STBY) /\ (M0ns STBY launch = ACTIVE) /\
    (M0ns ACTIVE reset = STBY) /\ (M0ns ACTIVE launch = ACTIVE)'

(* ----- *)
(* Define next-output function for machine M0 *)
(* ----- *)
val M0out_def =
  Define '(M0out STBY launch = on) /\ (M0out STBY reset = off) /\
    (M0out ACTIVE launch = on) /\ (M0out ACTIVE reset = off)'
```

The definitions above produce the defining theorems below.

```
[M0ns_def]
⊢ (M0ns STBY reset = STBY) ∧ (M0ns STBY launch = ACTIVE) ∧
  (M0ns ACTIVE reset = STBY) ∧ (M0ns ACTIVE launch = ACTIVE)
[M0out_def]
```

$$\vdash (M0out\ STBY\ launch = on) \wedge (M0out\ STBY\ reset = off) \wedge \\ (M0out\ ACTIVE\ launch = on) \wedge (M0out\ ACTIVE\ reset = off)$$

We see that each conjunct of  $M0ns$  and  $M0out$  corresponds to a row in the table defining  $M_0$  in Figure 16.3.

2. As usual, execute `Holmake` in a terminal in the subdirectory containing `m0Script.sml`. Make sure the theory compiles correctly in HOL.

After defining the next state and output functions, we prove properties of  $M_0$ . ◇

### Example 16.6

This example corresponds to the third step of our three-step process of defining state machines using *smTheory*. We use the inference rules defined in Section 16.1.7 to specialize the theorems in *smTheory* to machine  $M_0$ .

Do the following.

1. Add the following code to your script file. `m0Script.sml`.

```
(* ----- *)
(* Specialize TR_rules to m0 *)
(* ----- *)
val m0TR_rules = SPEC.TR ``M0ns`` ``M0out``

val _ = save_thm("m0TR_rules", m0TR_rules)

(* ----- *)
(* Specialize TR_clauses to m0 *)
(* ----- *)
val m0TR_clauses = SPEC.TR_clauses ``M0ns`` ``M0out``

val _ = save_thm("m0TR_clauses", m0TR_clauses)

(* ----- *)
(* Specialized Trans_Equiv_TR theorem to m0 *)
(* ----- *)
val m0Trans_Equiv_TR = SPEC.Trans_Equiv_TR ``M0ns`` ``M0out``

val _ = save_thm("m0Trans_Equiv_TR", m0Trans_Equiv_TR)
```

Notice that the above three theorems are proved using our custom inference rules `SPEC_TR`, `SPEC_TR_clauses`, and `SPEC_Trans_Equiv_TR`, which are defined in `sminfRules.sml`.

Three theorems are proved using the above code.

`[m0TR_rules]`

$$\vdash \forall s\ x\ ins\ outs. \\ TR\ x\ (CFG\ (x::ins)\ s\ outs) \\ (CFG\ ins\ (M0ns\ s\ x)\ (M0out\ s\ x::outs))$$

`[m0TR_clauses]`

$$\vdash (\forall x\ x1s\ s1\ out1s\ x2s\ out2s\ s2. \\ TR\ x\ (CFG\ x1s\ s1\ out1s)\ (CFG\ x2s\ s2\ out2s) \iff \\ \exists NS\ Out\ ins. \\ (x1s = x::ins) \wedge (x2s = ins) \wedge (s2 = NS\ s1\ x) \wedge$$

$$\begin{aligned}
& (out2s = Out\ s_1\ x::out1s) \wedge \\
& \forall x\ x1s\ s_1\ out1s\ x2s\ out2s. \\
& \quad TR\ x\ (CFG\ x1s\ s_1\ out1s) \\
& \quad (CFG\ x2s\ (M0ns\ s_1\ x)\ (M0out\ s_1\ x::out2s)) \iff \\
& \quad \exists ins. (x1s = x::ins) \wedge (x2s = ins) \wedge (out2s = out1s) \\
\text{[m0Trans\_Equiv\_TR]} \\
& \vdash TR\ x\ (CFG\ (x::ins)\ s\ outs) \\
& \quad (CFG\ ins\ (M0ns\ s\ x)\ (M0out\ s\ x::outs)) \iff \\
& \quad Trans\ x\ s\ (M0ns\ s\ x)
\end{aligned}$$

2. As usual, run `Holmake` within a terminal in the subdirectory containing `m0Script.sml`. Make sure that everything compiles correctly in HOL.

Additionally, we prove a theorem that corresponds to the tabular specification of  $M_0$  in Figure 16.3(b). Do the following.

1. Add the following code to `m0Script.sml`.

```

(* ----- *)
(* Theorems corresponding to the tabular specification of M0 *)
(* ----- *)
val th1 = REWRITE_RULE[M0ns_def, M0out_def](SPECL['STBY', 'launch'] m0TR_rules)
val th2 = REWRITE_RULE[M0ns_def, M0out_def](SPECL['STBY', 'reset'] m0TR_rules)
val th3 = REWRITE_RULE[M0ns_def, M0out_def](SPECL['ACTIVE', 'launch'] m0TR_rules)
val th4 = REWRITE_RULE[M0ns_def, M0out_def](SPECL['ACTIVE', 'reset'] m0TR_rules)

val m0_rules = LIST_CONJ [th1, th2, th3, th4]

val _ = save_thm("m0_rules", m0_rules)

```

The above specializes the *input* and current *state* corresponding to each row in Figure 16.3(b). Executing the code snippet above produces the following theorem, `m0_rules`.

$$\begin{aligned}
& \text{[m0\_rules]} \\
& \vdash (\forall ins\ outs. \\
& \quad TR\ launch\ (CFG\ (launch::ins)\ STBY\ outs) \\
& \quad (CFG\ ins\ ACTIVE\ (on::outs))) \wedge \\
& (\forall ins\ outs. \\
& \quad TR\ reset\ (CFG\ (reset::ins)\ STBY\ outs) \\
& \quad (CFG\ ins\ STBY\ (off::outs))) \wedge \\
& (\forall ins\ outs. \\
& \quad TR\ launch\ (CFG\ (launch::ins)\ ACTIVE\ outs) \\
& \quad (CFG\ ins\ ACTIVE\ (on::outs))) \wedge \\
& \forall ins\ outs. \\
& \quad TR\ reset\ (CFG\ (reset::ins)\ ACTIVE\ outs) \\
& \quad (CFG\ ins\ STBY\ (off::outs))
\end{aligned}$$

2. As usual, run `Holmake` within a terminal in the subdirectory containing `m0Script.sml`. Make sure that everything compiles correctly in HOL.

We are now done describing and formalizing  $M'_0$ 's behavior in HOL.

◇

**Figure 16.4** Tabular Description of a Simple Thermostat

command	Present State	Next State	Next Output
<i>Set k</i>	<i>State n</i>	<i>State k</i>	<i>DISPLAY k</i>
<i>Status</i>	<i>State n</i>	<i>State n</i>	<i>DISPLAY n</i>

### 16.2.2 Defining a Simple Thermostat

In this example, we define the state machine that is a simple thermostat whose behavior is specified in Figure 16.4. The tabular specification of behavior of the counter uses the following datatypes.

*command* = Set num | Status

*output* = DISPLAY num

*state* = State num

The following three examples show the three-step process of developing *simpleThermoTheory*: (1) defining the underlying datatypes, (2) defining the next-state and next-output functions, and (3) specializing the theorems of *smTheory* to the simple thermostat.

#### Example 16.7

In this example, we create the theory *simpleThermoTheory* and define the datatypes for *command*, *state*, *output* and their properties. Do the following.

1. As usual, create the file *simpleThermoScript.sml* and anticipate maintaining the theory using *Holmake*. The initial version of your file, before adding any definitions or proofs, should contain what is shown below. *simpleThermoTheory* depends on *TypeBase*, *smTheory*, and *sminfRules*.

```
structure simpleThermoScript = struct

open HolKernel boolLib Parse bossLib
open TypeBase smTheory sminfRules

val _ = new_theory "simpleThermo"

val _ = export_theory ()
val _ = print_theory "-"

end (* structure *)
```

2. Before adding anything else, run *Holmake* in a terminal in the same subdirectory in which *simpleThermoScript.sml* resides. Make sure, since *simpleThermoTheory* is built using *smTheory*, that your *Holmakefile* file has the path to *smTheory* assigned to *INCLUDES*.
3. Next, we define the datatypes for inputs, states, and outputs. We also prove the usual properties about them, namely that each element of each datatype is distinct from all the other members. Add the following code to *simpleThermoScript.sml*.

```
(*****
(* Define datatype for commands *)
(* *****)
val _ =
  Datatype 'command = Set num | Status '
(*****)
```

```

(* Prove distinctiveness and one to one properties of command *)
(*****)
val command_distinct_clauses = distinct_of '':command'
val _ = save_thm("command_distinct_clauses",command_distinct_clauses)

val command_one_one = one_one_of '':command'
val _ = save_thm("command_one_one",command_one_one)

```

The above code introduces the following datatype and distinctiveness and one-to-one theorems.

```

command = Set num | Status

[command_distinct_clauses]
 $\vdash \forall a. \text{Set } a \neq \text{Status}$ 

[command_one_one]
 $\vdash \forall a \ a'. (\text{Set } a = \text{Set } a') \iff (a = a')$ 

```

```

(*****)
(* Define thermostat state as a natural number corresponding to its setting *)
(*****)
val _ =
  Datatype 'state = State num'

(*****)
(* Prove one to one properties of state *)
(*****)
val state_one_one = one_one_of '':state'
val _ = save_thm("state_one_one",state_one_one)

```

The above code introduces the following datatype and one-to-one theorem.

```

state = State num

[state_one_one]
 $\vdash \forall a \ a'. (\text{State } a = \text{State } a') \iff (a = a')$ 

```

```

(*****)
(* Define thermostat output *)
(*****)
val _ =
  Datatype 'output = DISPLAY num'

(*****)
(* Prove one to one properties of output *)
(*****)
val output_one_one = one_one_of '':output'
val _ = save_thm("output_one_one",output_one_one)

```

The above code introduces the following datatype and one-to-one theorem.

```

output = DISPLAY num

[output_one_one]
 $\vdash \forall a \ a'. (\text{DISPLAY } a = \text{DISPLAY } a') \iff (a = a')$ 

```

4. Before adding anything else, run `Holmake` in a terminal in the same subdirectory in which `simpleThermoScript.sml` resides to make sure everything compiles correctly.  $\diamond$

**Example 16.8**

In this example, we define the next state and output functions for *simpleThermoTheory*, as defined in Figure 16.4. This is the second step on specializing *smTheory* to describe and define *simpleThermoTheory*. Do the following.

1. Add the following code to *simpleThermoScript.sml*.

```
(*****
(* Define simpleThermoNS, the next-state function *)
*****)
val simpleThermoNS_def =
Define
  '(simpleThermoNS (State n) (Set k) = (State k)) /\
  (simpleThermoNS (State n) (Status) = (State n))'

(*****
(* Define simpleThermoOut, the next-output function *)
*****)
val simpleThermoOut_def =
Define
  '(simpleThermoOut (State n) (Set k) = (DISPLAY k)) /\
  (simpleThermoOut (State n) Status = (DISPLAY n))'
```

The definitions above produce the defining theorems below.

```
[simpleThermoNS_def]
  ⊢ (simpleThermoNS (State n) (Set k) = State k) ∧
    (simpleThermoNS (State n) Status = State n)

[simpleThermoOut_def]
  ⊢ (simpleThermoOut (State n) (Set k) = DISPLAY k) ∧
    (simpleThermoOut (State n) Status = DISPLAY n)
```

We see that each conjunct of *simpleThermoNS* and *simpleThermoOut* corresponds to a row in the table defining the simple thermostat in Figure 16.4.

2. As usual, execute *Holmake* in a terminal in the subdirectory containing *simpleThermoScript.sml*. Make sure the theory compiles correctly in HOL.

After defining the next state and output functions, we prove properties of *simpleThermo*. ◇

**Example 16.9**

This example corresponds to the third step of our three-step process of defining state machines using *smTheory*. We use the inference rules defined in Section 16.1.7 to specialize the theorems in *smTheory* to *simpleThermoTheory*.

Do the following.

1. Add the following code to *simpleThermoScript.sml*.

```
(*****
(* Specialize TR_rules to simple thermostat *)
*****)
val simpleThermoTR_rules = SPEC.TR 'simpleThermoNS' 'simpleThermoOut'
val _ = save_thm("simpleThermoTR_rules", simpleThermoTR_rules)
```

```

(*****
(* Specialize TR_clauses to simple thermostat *)
(*****
val simpleThermoTR_clauses =
  SPEC_TR_clauses ``simpleThermoNS `` ``simpleThermoOut ``
val _ = save_thm("simpleThermoTR_clauses", simpleThermoTR_clauses)

(*****
(* Specialize Trans_Equiv_TR theorem to simple thermostat *)
(*****
val simpleThermoTrans_Equiv_TR =
  SPEC_Trans_Equiv_TR ``simpleThermoNS `` ``simpleThermoOut ``
val _ = save_thm("simpleThermoTrans_Equiv_TR", simpleThermoTrans_Equiv_TR)

```

Notice that the above three theorems are proved using our custom inference rules [SPEC\\_TR](#), [SPEC\\_TR\\_clauses](#), and [SPEC\\_Trans\\_Equiv\\_TR](#), which are defined in *sminfRules.sml*.

Three theorems are proved using the above code.

```

[simpleThermoTR\_rules]
⊢ ∀s x ins outs.
  TR x (CFG (x::ins) s outs)
    (CFG ins (simpleThermoNS s x)
      (simpleThermoOut s x::outs))

[simpleThermoTR\_clauses]
⊢ (∀x x1s s1 out1s x2s out2s s2.
  TR x (CFG x1s s1 out1s) (CFG x2s s2 out2s) ⇔
  ∃NS Out ins.
    (x1s = x::ins) ∧ (x2s = ins) ∧ (s2 = NS s1 x) ∧
    (out2s = Out s1 x::out1s)) ∧
  ∀x x1s s1 out1s x2s out2s.
  TR x (CFG x1s s1 out1s)
    (CFG x2s (simpleThermoNS s1 x)
      (simpleThermoOut s1 x::out2s)) ⇔
  ∃ins. (x1s = x::ins) ∧ (x2s = ins) ∧ (out2s = out1s))

[simpleThermoTrans\_Equiv\_TR]
⊢ TR x (CFG (x::ins) s outs)
  (CFG ins (simpleThermoNS s x)
    (simpleThermoOut s x::outs)) ⇔
  Trans x s (simpleThermoNS s x)

```

2. As usual, run `Holmake` within a terminal in the subdirectory containing *simpleThermoScript.sml*. Make sure that everything compiles correctly in HOL.

Additionally, we prove a theorem that corresponds to the tabular specification of the simple thermostat in Figure 16.4. Do the following.

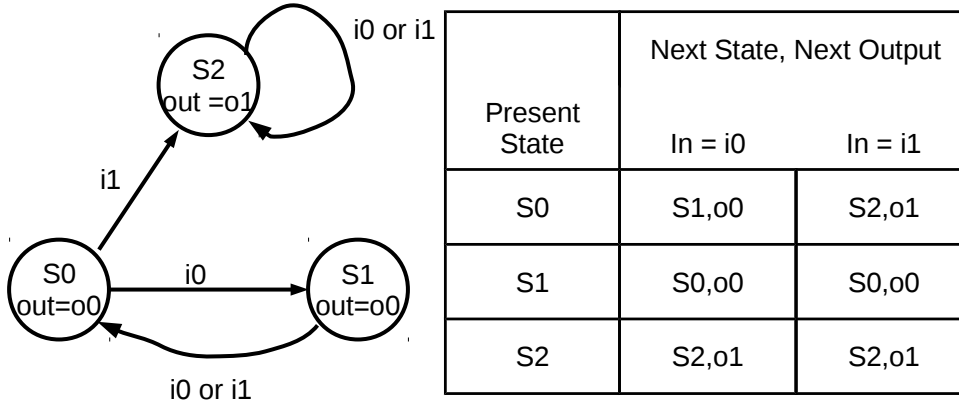
1. Add the following code to *simpleThermoScript.sml*.

```

(*****
(* Theorems corresponding to the tabular specification of the thermostat *)
(*****
val th1 =
  REWRITE_RULE

```



**Figure 16.5** Machine  $M_1$ 

```
[simpleThermoNS_def, simpleThermoOut_def]
(SPECL['State n', 'Set k'] simpleThermoTR_rules)
val th2 =
  REWRITE_RULE
    [simpleThermoNS_def, simpleThermoOut_def]
    (SPECL['State n', 'Status ''] simpleThermoTR_rules)
val simpleThermo_rules = LIST_CONJ [th1, th2]
val _ = save_thm("simpleThermo_rules", simpleThermo_rules)
```

The above specializes the *input* and current *state* corresponding to each row in Figure 16.4. Executing the code snippet above produces the following theorem, *simpleThermoTR\_rules*.

```
[simpleThermo_rules]
⊢ (∀ins outs.
  TR (Set k) (CFG (Set k::ins) (State n) outs)
    (CFG ins (State k) (DISPLAY k::outs))) ∧
  ∀ins outs.
    TR Status (CFG (Status::ins) (State n) outs)
      (CFG ins (State n) (DISPLAY n::outs))
```

- As usual, run Holmake within a terminal in the subdirectory containing *simpleThermoScript.sml*. Make sure that everything compiles correctly in HOL.

We are now done describing and formalizing the simple thermostat in HOL. ◇

## 16.3 Exercises

**Exercise 16.3.1** Consider the machine  $M_1$  pictured in Figure 16.5. Your task is to define a theory *m1Theory* for machine  $M_1$  in exactly the same way as illustrated in Examples 16.4, 16.5, and 16.6. Do the following.

- Define the datatypes for inputs, states, and outputs; prove theorems about the datatypes. In particular, introduce the following datatype and their theorems. **Make sure you use the same names for types and theorems as shown below.**

```
command = i0 | i1
```

```

state = S0 | S1 | S2
output = o0 | o1

[command_distinct_clauses]
⊢ i0 ≠ i1

[state_distinct_clauses]
⊢ S0 ≠ S1 ∧ S0 ≠ S2 ∧ S1 ≠ S2

[output_distinct_clauses]
⊢ o0 ≠ o1

```

B. Define the next state and next output functions for  $M_1$ . **Make sure you use the names  $M1ns$  and  $M1out$  as the function names.**

C. Prove the following theorems about  $M_1$ . **Make sure you use the same names for theorems as shown below.**

```

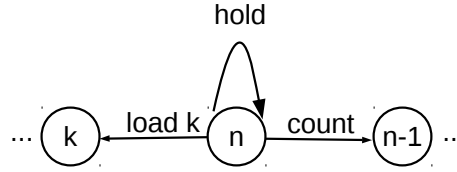
[m1TR_rules]
⊢ ∀s x ins outs.
  TR x (CFG (x::ins) s outs)
    (CFG ins (M1ns s x) (M1out s x::outs))

[m1TR_clauses]
⊢ (∀x x1s s1 out1s x2s out2s s2.
  TR x (CFG x1s s1 out1s) (CFG x2s s2 out2s) ⇔
  ∃NS Out ins.
    (x1s = x::ins) ∧ (x2s = ins) ∧ (s2 = NS s1 x) ∧
    (out2s = Out s1 x::out1s)) ∧
  ∀x x1s s1 out1s x2s out2s.
  TR x (CFG x1s s1 out1s)
    (CFG x2s (M1ns s1 x) (M1out s1 x::out2s)) ⇔
  ∃ins. (x1s = x::ins) ∧ (x2s = ins) ∧ (out2s = out1s))

[m1Trans_Equiv_TR]
⊢ TR x (CFG (x::ins) s outs)
  (CFG ins (M1ns s x) (M1out s x::outs)) ⇔
  Trans x s (M1ns s x)

[m1_rules]
⊢ (∀ins outs.
  TR i0 (CFG (i0::ins) S0 outs) (CFG ins S1 (o0::outs))) ∧
  (∀ins outs.
  TR i1 (CFG (i1::ins) S0 outs) (CFG ins S2 (o1::outs))) ∧
  (∀ins outs.
  TR i0 (CFG (i0::ins) S1 outs) (CFG ins S0 (o0::outs))) ∧
  (∀ins outs.
  TR i1 (CFG (i1::ins) S1 outs) (CFG ins S0 (o0::outs))) ∧
  (∀ins outs.
  TR i0 (CFG (i0::ins) S2 outs) (CFG ins S2 (o1::outs))) ∧
  (∀ins outs.
  TR i1 (CFG (i1::ins) S2 outs) (CFG ins S2 (o1::outs)))

```

**Figure 16.7** Partial State-Transition Diagram for Down Counter**Exercise 16.3.2****Figure 16.6** Tabular Description of Down-Counter Behavior

command	Present State	Next State	Next Output
<i>load k</i>	<i>COUNT n</i>	<i>COUNT k</i>	<i>DISPLAY k</i>
<i>count</i>	<i>COUNT n</i>	<i>COUNT (n - 1)</i> Note: $0 - k = 0$ for subtraction on natural numbers	<i>DISPLAY (n - 1)</i>
<i>hold</i>	<i>COUNT n</i>	<i>COUNT n</i>	<i>DISPLAY n</i>

Consider the down counter shown in Figures 16.6 and 16.7. Your task is to define a theory counterTheory in exactly the same way as illustrated in Examples 16.7, 16.8, and 16.9. Do the following.

- A. Define the datatypes for inputs, states, and outputs; prove theorems about the datatypes. In particular, introduce the following datatype and their theorems. **Make sure you use the same names for types and theorems as shown below.**

```

ctrCmd = load num | count | hold
ctrState = COUNT num
ctrOut = DISPLAY num

[ctrCmd_distinct_clauses]
⊢ (∀a. load a ≠ count) ∧ (∀a. load a ≠ hold) ∧ count ≠ hold

[ctrState_one_one]
⊢ ∀a a'. (COUNT a = COUNT a') ⇔ (a = a')

[ctrOut_one_one]
⊢ ∀a a'. (DISPLAY a = DISPLAY a') ⇔ (a = a')

```

- B. Define the next state and next output functions for counterTheory. **Make sure you use the names ctrNS and ctrOut as the function names.**

- C. Prove the following theorems about the counter. **Make sure you use the same names for theorems as shown below.**

```

[ctrTR_rules]
⊢ ∀s x ins outs.
  TR x (CFG (x::ins) s outs)
    (CFG ins (ctrNS s x) (ctrOut s x::outs))

[ctrTR_clauses]

```

$$\begin{aligned}
&\vdash (\forall x \ x1s \ s_1 \ out1s \ x2s \ out2s \ s_2. \\
&\quad TR \ x \ (CFG \ x1s \ s_1 \ out1s) \ (CFG \ x2s \ s_2 \ out2s) \iff \\
&\quad \exists NS \ Out \ ins. \\
&\quad \quad (x1s = x::ins) \wedge (x2s = ins) \wedge (s_2 = NS \ s_1 \ x) \wedge \\
&\quad \quad (out2s = Out \ s_1 \ x::out1s)) \wedge \\
&\forall x \ x1s \ s_1 \ out1s \ x2s \ out2s. \\
&\quad TR \ x \ (CFG \ x1s \ s_1 \ out1s) \\
&\quad \quad (CFG \ x2s \ (ctrNS \ s_1 \ x) \ (ctrOut \ s_1 \ x::out2s)) \iff \\
&\quad \exists ins. (x1s = x::ins) \wedge (x2s = ins) \wedge (out2s = out1s)
\end{aligned}$$

*[ctrTrans\_Equiv\_TR]*

$$\begin{aligned}
&\vdash TR \ x \ (CFG \ (x::ins) \ s \ outs) \\
&\quad (CFG \ ins \ (ctrNS \ s \ x) \ (ctrOut \ s \ x::outs)) \iff \\
&\quad Trans \ x \ s \ (ctrNS \ s \ x)
\end{aligned}$$

*[ctr\_rules]*

$$\begin{aligned}
&\vdash (\forall ins \ outs. \\
&\quad TR \ (load \ new) \ (CFG \ (load \ new::ins) \ (COUNT \ n) \ outs) \\
&\quad \quad (CFG \ ins \ (COUNT \ new) \ (DISPLAY \ new::outs)) \wedge \\
&\forall ins \ outs. \\
&\quad TR \ count \ (CFG \ (count::ins) \ (COUNT \ n) \ outs) \\
&\quad \quad (CFG \ ins \ (COUNT \ (n - 1)) \ (DISPLAY \ (n - 1)::outs)) \wedge \\
&\forall ins \ outs. \\
&\quad TR \ hold \ (CFG \ (hold::ins) \ (COUNT \ n) \ outs) \\
&\quad \quad (CFG \ ins \ (COUNT \ n) \ (DISPLAY \ n::outs))
\end{aligned}$$

# Secure State Machines

In Chapter 16, we developed formal descriptions of behavior of state machines using *configurations* and *labeled transition relations*, where next state and output functions are parameters of state-machine configurations. By taking advantage of higher-order logic's ability to parameterize over functions, the inductive definitions of transition relations *Trans* and *TR* are general definitions capable of being specialized to any particular synchronous state-machine by specifying

1. A set of states  $S = \{s_0, \dots, s_{n-1}, \dots\}$ ,
2. A set of inputs  $I = \{i_0, \dots, i_k, \dots\}$ ,
3. A set of outputs  $O = \{o_0, \dots, o_j, \dots\}$ ,
4. A next-state transition function on  $\delta : S \rightarrow I \rightarrow S$ , and
5. An output function  $\lambda : S \rightarrow I \rightarrow O$ , which is used to compute the output in the next clock period.

In this chapter, we develop formal descriptions of *secure state machines*, which have the parameters above augmented with the parameters below:

1. An input authentication function that checks the integrity and origin of an input command,
2. A state interpretation function that interprets a machine's state in the access-control logic, e.g., if a machine is operating in a privileged mode or not, and
3. A security context that authorizes or traps authenticated input commands, where the context is given by a list of certificates, trust assumptions, policy statements, delegations, and authorizations.

## 17.1 A High-Level Secure State-Machine

In this section, we introduce the theory *ssmlTheory*, which is a parameterized definition of a secure state-machine with the following parameters:

1. States are parameterized as polymorphic type variables `'state`,
2. Inputs to the secure state-machine are represented as access-control logic formulas with type

`('command inst, 'principal,  $\delta$ ,  $\epsilon$ ) Form,`

where the `'command inst` type is defined later in this section,

3. Outputs of the secure state-machine are parameterized as polymorphic type variables `'output`,

4. A next-state function ( $NS : 'state \rightarrow 'command \ trType \rightarrow 'state$ ), where the type  $'command \ trType$  specifies if the state transition is due to discarding, trapping, or executing a command, and is defined later in this section,
5. An output function ( $Out : 'state \rightarrow 'command \ trType \rightarrow 'output$ ),
6. An input testing function

$(inputTest : ('command \ inst, 'principal, \delta, \epsilon) \text{Form} \rightarrow \text{bool})$ ,

which checks the integrity and authenticity of input commands,

7. A state-interpretation function

$(stateInterp : 'state \rightarrow ('command \ inst, 'principal, \delta, \epsilon) \text{Form})$ ,

and

8. A security context given by a list of certificates, delegations, trust assumptions, and policies described by access-control logic formulas,

$(certList : ('command \ inst, 'principal, \delta, \epsilon) \text{Form list})$ .

### 17.1.1 Building and Loading *ssm1Theory*

Five source files are required to build *ssm1Theory*. These files are

1. *Holmakefile*, which specifies additional subdirectories for HOL to use when loading component theories. Note the use of the variable  $\${HOME}$  in the path specifying the subdirectory containing the access-control logic theory files. The variable  $\${HOME}$  is the path to user's home directory.
2. *satListScript.sml*, which defines a relation `satList`. Essentially, what  $(M, Oi, Os) \text{ satList } [f_0; \dots; f_{n-1}]$  returns is

$$(M, Oi, Os) \models f_0 \wedge \dots \wedge (M, Oi, Os) \models f_{n-1}$$

The above is verified by the three theorems below.

`[satList_nil]`

$\vdash (M, Oi, Os) \text{ satList } []$

`[satList_CONS]`

$\vdash (M, Oi, Os) \text{ satList } (h :: t) \iff$   
 $(M, Oi, Os) \text{ sat } h \wedge (M, Oi, Os) \text{ satList } t$

`[satList_conj]`

$\vdash (M, Oi, Os) \text{ satList } l_1 \wedge (M, Oi, Os) \text{ satList } l_2 \iff$   
 $(M, Oi, Os) \text{ satList } (l_1 ++ l_2)$

3. *ssminfRules.sml*, which defines three ML functions `flip_imp`, `flip_TR_rules`, and `TR_EQ_rules` used to prove theorems in *ssm1Theory*.

4. *ssminfRules.sig*, which lists the type signatures needed by ML to make the functions `flip_imp`, `flip_TR_rules`, and `TR_EQ_rules` loadable by HOL.
5. *ssm1Script.sml*, which defines *ssm1Theory*.

The source code for all the files above is in Section 17.4. Placing all of the files in the same subdirectory and executing `Holmake cleanAll` followed by `Holmake` will build *ssm1Theory*. The theory is loaded and opened in HOL as usual by `load "ssm1Theory"` and `open ssm1Theory`.

### 17.1.2 Basic Types Used by *ssm1Theory*

*ssm1Theory* introduces three new types.

1. The type `'command inst`, which takes state-machine inputs, given by the type variable `'command`, and uses the HOL *option* type to add `NONE` to the inputs. `'command inst` is defined as follows.

$$inst = \text{SOME } 'command \mid \text{NONE}$$

Recall, *option* types are used in modeling the idealized behavior of cryptographic function in HOL, as described in Section 15.3.2. All the elements of `'command` are present in `'command inst` with the addition of `NONE`. We use `NONE` to indicate when an instruction should be *trapped*.

The typical distinctiveness and one-to-one properties of `'command inst` are proved using *Type-Base* functions `distinct_of` and `one_one_of`.

`[inst_distinct_clauses]`

$$\vdash \text{SOME } a \neq \text{NONE}$$

`[inst_one_one]`

$$\vdash (\text{SOME } a = \text{SOME } a') \iff (a = a')$$

2. The type `'command trType`, which is used to label the three kinds of transitions TR secure state-machines in *sm1Theory* can make

- (a) `discard` transitions, which occur when input (`cmd : 'command`) fails to be authenticated or deemed intact,
- (b) `trap cmd` transitions, which occur when input (`cmd : 'command`) is authenticated but unauthorized, and
- (c) `exec cmd` transitions, which occur when input `cmd` is both authenticated and authorized.

The definition of `'command trType` is as follows.

$$trType = \text{discard} \mid \text{trap } 'command \mid \text{exec } 'command$$

The typical distinctiveness and one-to-one properties are proved.

`[trType_distinct_clauses]`

$$\vdash (\forall a. \text{discard} \neq \text{trap } a) \wedge (\forall a. \text{discard} \neq \text{exec } a) \wedge \forall a' a. \text{trap } a \neq \text{exec } a'$$

```
[trType_one_one]
⊢ (∀a a'. (trap a = trap a') ⇔ (a = a')) ∧
  ∀a a'. (exec a = exec a') ⇔ (a = a')
```

3. The type  $(\text{'command}, \delta, \epsilon, \text{'output}, \text{'principal}, \text{'state})$  configuration defines the *configurations* of secure state-machines. Its definition is as follows.

```
configuration =
  CFG (('command inst, 'principal, 'd, 'e) Form -> bool)
      (('state -> ('command inst, 'principal, 'd, 'e) Form)
       (('command inst, 'principal, 'd, 'e) Form list)
       (('command inst, 'principal, 'd, 'e) Form list) 'state
       ('output list))
```

The *configuration* parameters are listed below in order in which they appear.

- (a) An authentication function

```
(inputTest : ('command inst, 'principal, δ, ε) Form -> bool)
```

- (b) A state interpretation function

```
(stateInterp : 'state -> ('command inst, 'principal, δ, ε) Form)
```

- (c) A security context given as a list of access-control logic formulas

```
(certList : ('command inst, 'principal, δ, ε) Form list)
```

- (d) A list of input commands

```
(ins : ('command inst, 'principal, δ, ε) Form list)
```

- (e) A state

```
(s : 'state)
```

- (f) A list of outputs

```
(outs : 'output list)
```

The one-to-one property of *configuration* is proved.

```
[configuration_one_one]
⊢ (CFG a0 a1 a2 a3 a4 a5 = CFG a'0 a'1 a'2 a'3 a'4 a'5) ⇔
  (a0 = a'0) ∧ (a1 = a'1) ∧ (a2 = a'2) ∧ (a3 = a'3) ∧
  (a4 = a'4) ∧ (a5 = a'5)
```

### 17.1.3 Interpreting *configurations* in the Access-Control Logic

The function `CFGInterpret` gives the meaning of a particular *configuration* in the access-control logic. Essentially, the meaning of a *configuration* is the meaning in the access-control logic of its security *context*, *state*, and current input *x*. The definition of `CFGInterpret` is as follows.

```
[CFGInterpret_def]
⊢ CFGInterpret (M, Oi, Os)
  (CFG inputTest stateInterp context (x::ins) state
   outStream) ⇔
  (M, Oi, Os) satList context ∧ (M, Oi, Os) sat x ∧
  (M, Oi, Os) sat stateInterp state
```



### 17.1.4 Defining the Configuration Transition Relation TR

The labeled-transition relation TR is defined using 'command trType, defined as

$$trType = \text{discard} \mid \text{trap 'command} \mid \text{exec 'command}$$

There are three kinds of TR transitions, (1) `exec cmd`, (2) `trap cmd`, and (3) `discard`, which correspond to executing a command, trapping it, or ignoring it, respectively.

1. In words, the transition rule for executing commands is if:

- (a) the input  $P$  says `prop (SOME cmd)` passes the integrity check `inputTest`, and
- (b) the meaning of the starting configuration

$$\text{CFG } inputTest \ stateInterp \ certList \ (P \ \text{says} \ \text{prop} \ (\text{SOME } cmd) :: ins) \ s \ outs$$

is given by  $\text{CFGInterpret } (M, Oi, Os)$ ,

then the starting configuration can make a TR  $(M, Oi, Os) \ (\text{exec } cmd)$  transition to the next configuration

$$\text{CFG } inputTest \ stateInterp \ certList \ ins \ (NS \ s \ (\text{exec } cmd)) \ (Out \ s \ (\text{exec } cmd) :: outs).$$

This is symbolized using  $\xrightarrow{\text{exec } cmd}$  to represent TR  $(M, Oi, Os) \ (\text{exec } cmd)$  as follows.

$$\frac{\text{CFGInterpret } (M, Oi, Os) \ (\text{CFG } inputTest \ stateInterp \ certList \ (P \ \text{says} \ \text{prop} \ (\text{SOME } cmd) :: ins) \ s \ outs) \quad inputTest \ (P \ \text{says} \ \text{prop} \ (\text{SOME } cmd))}{(\text{CFG } inputTest \ stateInterp \ certList \ ((P \ \text{says} \ \text{prop} \ (\text{SOME } cmd)) :: ins) \ s \ outs) \xrightarrow{\text{exec } cmd} (\text{CFG } inputTest \ stateInterp \ certList \ ins \ (NS \ s \ (\text{exec } cmd)) \ (Out \ s \ (\text{exec } cmd) :: outs))}$$

The corresponding theorem in HOL is

$$\begin{aligned} & \vdash inputTest \ (P \ \text{says} \ \text{prop} \ (\text{SOME } cmd)) \ \wedge \\ & \text{CFGInterpret } (M, Oi, Os) \\ & \quad (\text{CFG } inputTest \ stateInterp \ certList \\ & \quad \quad (P \ \text{says} \ \text{prop} \ (\text{SOME } cmd) :: ins) \ s \ outs) \Rightarrow \\ & \text{TR } (M, Oi, Os) \ (\text{exec } cmd) \\ & \quad (\text{CFG } inputTest \ stateInterp \ certList \\ & \quad \quad (P \ \text{says} \ \text{prop} \ (\text{SOME } cmd) :: ins) \ s \ outs) \\ & \quad (\text{CFG } inputTest \ stateInterp \ certList \ ins \ (NS \ s \ (\text{exec } cmd)) \\ & \quad \quad (Out \ s \ (\text{exec } cmd) :: outs)) \end{aligned}$$

2. Similar to executing commands, the transition rule for trapping commands is

- (a) the input  $P$  says `prop (SOME cmd)` passes the integrity check `inputTest`, and
- (b) the meaning of the starting configuration

$$\text{CFG } inputTest \ stateInterp \ certList \ (P \ \text{says} \ \text{prop} \ (\text{SOME } cmd) :: ins) \ s \ outs$$

is given by  $\text{CFGInterpret } (M, Oi, Os)$ ,

then the starting configuration can make a  $\text{TR } (M, Oi, Os) \text{ (trap cmd)}$  transition to the next configuration

$\text{CFG inputTest stateInterp certList ins (NS s (trap cmd)) (Out s (trap cmd)::outs)}$ .

This is symbolized using  $\xrightarrow{\text{trap cmd}}$  to represent  $\text{TR } (M, Oi, Os) \text{ (trap cmd)}$  as follows.

$$\frac{\text{CFGInterpret } (M, Oi, Os) \text{ (CFG inputTest stateInterp certList (P says prop (SOME cmd)::ins) s outs) } \quad \text{inputTest (P says prop (SOME cmd))}}{\text{(CFG inputTest stateInterp certList ins (NS s (trap cmd)) (Out s (trap cmd)::outs))} \xrightarrow{\text{trap cmd}} \text{(CFG inputTest stateInterp certList (P says prop (SOME cmd)::ins) s outs)}}$$

The corresponding theorem in HOL is

$$\begin{aligned} & \vdash \text{inputTest (P says prop (SOME cmd))} \wedge \\ & \text{CFGInterpret (M, Oi, Os)} \\ & \quad (\text{CFG inputTest stateInterp certList} \\ & \quad \quad (\text{P says prop (SOME cmd)::ins) s outs}) \Rightarrow \\ & \text{TR (M, Oi, Os) (trap cmd)} \\ & \quad (\text{CFG inputTest stateInterp certList} \\ & \quad \quad (\text{P says prop (SOME cmd)::ins) s outs}) \\ & \quad (\text{CFG inputTest stateInterp certList ins (NS s (trap cmd))} \\ & \quad \quad (\text{Out s (trap cmd)::outs})) \end{aligned}$$

3. Finally, the transition rule for discarding commands is if the integrity check *inputTest* fails on input *x*, then *x* is discarded. The rule is symbolized as follows.

$$\frac{\neg \text{inputTest } x}{\text{(CFG inputTest stateInterp certList (x::ins) s outs) \xrightarrow{\text{discard}} \text{(CFG inputTest stateInterp certList ins (NS s discard) (Out s discard::outs))}}}$$

The corresponding theorem in HOL is

$$\begin{aligned} & \vdash \neg \text{inputTest } x \Rightarrow \\ & \text{TR (M, Oi, Os) discard} \\ & \quad (\text{CFG inputTest stateInterp certList (x::ins) s outs}) \\ & \quad (\text{CFG inputTest stateInterp certList ins (NS s discard)} \\ & \quad \quad (\text{Out s discard::outs})) \end{aligned}$$

The three theorems characterizing the inductive relation *TR* are defined and proved using *Hol\_reln*, as normal. This results in the three theorems [TR\\_rules](#), [TR\\_ind](#), and [TR\\_cases](#)

### 17.1.5 Equality Theorems for TR

Based on the above definitions of TR and CFGInterpret, we have three equivalence rules, *TR\_discard\_cmd\_rule*, *TR\_trap\_cmd\_rule*, and *TR\_exec\_cmd\_rule*. We describe each of these theorems below.

**TR\_discard\_cmd\_rule** This theorem states that a *discard* transition occurs if and only if  $\neg \text{inputTest } x$  is true, i.e., if the input  $x$  fails its integrity check. Notice that the bi-conditional form of this rule assures that a failed integrity check means unauthenticated inputs are discarded, and discarded inputs are those that failed integrity checking as defined by parameter *inputTest*.

$$\begin{aligned} & (\text{CFG } \text{inputTest } \text{stateInterp } \text{certList } (x :: \text{ins}) \ s \ \text{outs}) \xrightarrow{\text{discard}} \\ & (\text{CFG } \text{inputTest } \text{stateInterp } \text{certList } \text{ins } (\text{NS } s \ \text{discard}) \ (\text{Out } s \ \text{discard} :: \text{outs})) \\ & \iff \\ & \neg \text{inputTest } x \end{aligned}$$

The HOL theorem is as follows.

[TR\_discard\_cmd\_rule]

$\vdash \text{TR } (M, Oi, Os) \ \text{discard}$   
 $(\text{CFG } \text{inputTest } \text{stateInterp } \text{certList } (x :: \text{ins}) \ s \ \text{outs})$   
 $(\text{CFG } \text{inputTest } \text{stateInterp } \text{certList } \text{ins } (\text{NS } s \ \text{discard})$   
 $(\text{Out } s \ \text{discard} :: \text{outs})) \iff \neg \text{inputTest } x$

**TR\_trap\_cmd\_rule** This rule states

1. **If** the following is true, i.e., is a derived inference rule in the access-control logic

$$\frac{\text{CFGInterpret } (M, Oi, Os) \ (\text{CFG } \text{inputTest } \text{stateInterp } \text{certList } (P \ \text{says prop (SOME cmd)} :: \text{ins}) \ s \ \text{outs})}{(M, Oi, Os) \ \text{sat prop NONE}}$$

2. **then** the following bi-conditional holds

$$\begin{aligned} & (\text{CFG } \text{inputTest } \text{stateInterp } \text{certList } (x :: \text{ins}) \ s \ \text{outs}) \xrightarrow{\text{trap cmd}} \\ & (\text{CFG } \text{inputTest } \text{stateInterp } \text{certList } \text{ins } (\text{NS } s \ (\text{trap cmd})) \ (\text{Out } s \ (\text{trap cmd}) :: \text{outs})) \\ & \iff \\ & (\text{inputTest } (P \ \text{says prop (SOME cmd)}) \wedge \\ & \quad \text{CFGInterpret } (M, Oi, Os) \\ & \quad \text{CFG } \text{inputTest } \text{stateInterp } \text{certList } (P \ \text{says prop (SOME cmd)} :: \text{ins}) \ s \ \text{outs} \wedge \\ & \quad (M, Oi, Os) \ \text{sat prop NONE}) \end{aligned}$$

In other words, if *prop NONE* is derivable from the interpretation of the starting configuration, then we know that a *trap cmd* transition occurs if and only if

1. the input  $P \ \text{says prop (SOME cmd)}$  is authenticated,
2. we are interpreting the starting configuration using *CFGInterpret*, and
3. *prop NONE* is true, i.e. the command *cmd* should be trapped.

The HOL theorem is as follows.

```

[TR_trap_cmd_rule]
⊢ (∀M Oi Os.
  CFGInterpret (M, Oi, Os)
    (CFG inputTest stateInterp certList
      (P says prop (SOME cmd) :: ins) s outs) ⇒
    (M, Oi, Os) sat prop NONE) ⇒
  ∀NS Out M Oi Os.
    TR (M, Oi, Os) (trap cmd)
      (CFG inputTest stateInterp certList
        (P says prop (SOME cmd) :: ins) s outs)
      (CFG inputTest stateInterp certList ins
        (NS s (trap cmd)) (Out s (trap cmd) :: outs)) ⇔
    inputTest (P says prop (SOME cmd)) ∧
    CFGInterpret (M, Oi, Os)
      (CFG inputTest stateInterp certList
        (P says prop (SOME cmd) :: ins) s outs) ∧
    (M, Oi, Os) sat prop NONE

```

**TR.exec.cmd.rule** This rule states

1. **If** the following is true, i.e., is a derived inference rule in the access-control logic

$$\frac{\text{CFGInterpret } (M, Oi, Os) \text{ (CFG inputTest stateInterp certList (P says prop (SOME cmd) :: ins) s outs)}}{(M, Oi, Os) \text{ sat prop (SOME cmd)}}$$

2. **then** the following bi-conditional holds

$$\begin{aligned} & (\text{CFG inputTest stateInterp certList } (x :: \text{ins}) \text{ s outs}) \xrightarrow{\text{exec cmd}} \\ & (\text{CFG inputTest stateInterp certList ins (NS s (exec cmd)) (Out s (exec cmd) :: outs)}) \\ & \iff \\ & (\text{inputTest (P says prop (SOME cmd))} \wedge \\ & \quad \text{CFGInterpret (M, Oi, Os)} \\ & \quad \text{CFG inputTest stateInterp certList (P says prop (SOME cmd) :: ins) s outs} \wedge \\ & \quad (M, Oi, Os) \text{ sat prop (SOME cmd)}) \end{aligned}$$

In other words, if  $\text{prop (SOME cmd)}$  is derivable from the interpretation of the starting configuration, then we know that an  $\text{exec cmd}$  transition occurs if and only if

1. the input  $P \text{ says prop (SOME cmd)}$  is authenticated,
2. we are interpreting the starting configuration using  $\text{CFGInterpret}$ , and
3.  $\text{prop (SOME cmd)}$  is true, i.e. the command  $\text{cmd}$  should be executed.

The HOL theorem is as follows.

```

[TR_exec_cmd_rule]
⊢ (∀M Oi Os.
  CFGInterpret (M, Oi, Os)
    (CFG inputTest stateInterp certList

```

$$\begin{aligned}
& (P \text{ says prop (SOME cmd) :: ins) } s \text{ outs} \Rightarrow \\
& (M, Oi, Os) \text{ sat prop (SOME cmd)} \Rightarrow \\
& \forall NS \text{ Out } M \text{ Oi } Os. \\
& \text{TR } (M, Oi, Os) \text{ (exec cmd)} \\
& \quad (\text{CFG inputTest stateInterp certList} \\
& \quad \quad (P \text{ says prop (SOME cmd) :: ins) } s \text{ outs}) \\
& \quad (\text{CFG inputTest stateInterp certList ins} \\
& \quad \quad (NS \text{ s (exec cmd)}) \text{ (Out s (exec cmd) :: outs)}) \iff \\
& \text{inputTest } (P \text{ says prop (SOME cmd)}) \wedge \\
& \text{CFGInterpret } (M, Oi, Os) \\
& \quad (\text{CFG inputTest stateInterp certList} \\
& \quad \quad (P \text{ says prop (SOME cmd) :: ins) } s \text{ outs}) \wedge \\
& (M, Oi, Os) \text{ sat prop (SOME cmd)}
\end{aligned}$$

## 17.2 Defining Secure State-Machine $SM_0$ Using *ssm1Theory*

For our first example applying *ssm1Theory*, we develop a secure version of machine  $M_0$ , that appeared in Section 16.2.1. We call the secure version of the machine  $SM_0$ .

### 17.2.1 $SM_0$ Datatypes

The code snippets below introduce the following datatypes and their properties:

1. *privcmd* = launch | reset

```

val _ =
  Datatype 'privcmd = launch | reset '

val privcmd_distinct_clauses = distinct_of '':privcmd''
val _ = save_thm("privcmd_distinct_clauses", privcmd_distinct_clauses)

```

```

[privcmd_distinct_clauses]
⊢ launch ≠ reset

```

2. *npriv* = status

```

val _ =
  Datatype 'npriv = status '

```

3. *command* = NP *npriv* | PR *privcmd*

```

val _ =
  Datatype 'command = NP npriv | PR privcmd '

val command_distinct_clauses = distinct_of '':command''
val _ = save_thm("command_distinct_clauses", command_distinct_clauses)

val command_one_one = one_one_of '':command''
val _ = save_thm("command_one_one", command_one_one)

```

[command\_distinct\_clauses]

$\vdash \text{NP } a \neq \text{PR } a'$

[command\_one\_one]

$\vdash (\forall a \ a'. \ (\text{NP } a = \text{NP } a') \iff (a = a')) \wedge$   
 $\forall a \ a'. \ (\text{PR } a = \text{PR } a') \iff (a = a')$

4. *state* = STBY | ACTIVE

```
val _ =
  Datatype 'state = STBY | ACTIVE'

val state_distinct_clauses = distinct_of '':state''
val _ = save_thm("state_distinct_clauses", state_distinct_clauses)
```

[state\_distinct\_clauses]

$\vdash \text{STBY} \neq \text{ACTIVE}$

5. *output* = on | off

```
val _ =
  Datatype 'output = on | off'

val output_distinct_clauses = distinct_of '':output''
val _ = save_thm("output_distinct_clauses", output_distinct_clauses)
```

[output\_distinct\_clauses]

$\vdash \text{on} \neq \text{off}$

6. *staff* = Alice | Bob | Carol

```
val _ =
  Datatype 'staff = Alice | Bob | Carol'

val staff_distinct_clauses = distinct_of '':staff''
val _ = save_thm("staff_distinct_clauses", staff_distinct_clauses)
```

[staff\_distinct\_clauses]

$\vdash \text{Alice} \neq \text{Bob} \wedge \text{Alice} \neq \text{Carol} \wedge \text{Bob} \neq \text{Carol}$

### 17.2.2 $SM_0$ Next-State Function

The following code snippet defines the next-state function for  $SM_0$ .

```

val SM0ns_def =
Define
‘(SM0ns STBY (exec (PR reset)) = STBY) /\
(SM0ns STBY (exec (PR launch)) = ACTIVE) /\
(SM0ns STBY (exec (NP status)) = STBY) /\
(SM0ns ACTIVE (exec (PR reset)) = STBY) /\
(SM0ns ACTIVE (exec (PR launch)) = ACTIVE) /\
(SM0ns ACTIVE (exec (NP status)) = ACTIVE) /\
(SM0ns STBY (trap (PR reset)) = STBY) /\
(SM0ns STBY (trap (PR launch)) = STBY) /\
(SM0ns STBY (trap (NP status)) = STBY) /\
(SM0ns ACTIVE (trap (PR reset)) = ACTIVE) /\
(SM0ns ACTIVE (trap (PR launch)) = ACTIVE) /\
(SM0ns ACTIVE (trap (NP status)) = ACTIVE) /\
(SM0ns STBY discard = STBY) /\
(SM0ns ACTIVE discard = ACTIVE)’

```

[SM0ns\_def]

```

⊢ (SM0ns STBY (exec (PR reset)) = STBY) ∧
  (SM0ns STBY (exec (PR launch)) = ACTIVE) ∧
  (SM0ns STBY (exec (NP status)) = STBY) ∧
  (SM0ns ACTIVE (exec (PR reset)) = STBY) ∧
  (SM0ns ACTIVE (exec (PR launch)) = ACTIVE) ∧
  (SM0ns ACTIVE (exec (NP status)) = ACTIVE) ∧
  (SM0ns STBY (trap (PR reset)) = STBY) ∧
  (SM0ns STBY (trap (PR launch)) = STBY) ∧
  (SM0ns STBY (trap (NP status)) = STBY) ∧
  (SM0ns ACTIVE (trap (PR reset)) = ACTIVE) ∧
  (SM0ns ACTIVE (trap (PR launch)) = ACTIVE) ∧
  (SM0ns ACTIVE (trap (NP status)) = ACTIVE) ∧
  (SM0ns STBY discard = STBY) ∧
  (SM0ns ACTIVE discard = ACTIVE)

```

### 17.2.3 $SM_0$ Next-Output Function

The following code snippet defines the next-output function for  $SM_0$ .

```

val SM0out_def =
Define
‘(SM0out STBY (exec (PR reset)) = off) /\
(SM0out STBY (exec (PR launch)) = on) /\
(SM0out STBY (exec (NP status)) = off) /\
(SM0out ACTIVE (exec (PR reset)) = off) /\
(SM0out ACTIVE (exec (PR launch)) = on) /\
(SM0out ACTIVE (exec (NP status)) = on) /\
(SM0out STBY (trap (PR reset)) = off) /\
(SM0out STBY (trap (PR launch)) = off) /\
(SM0out STBY (trap (NP status)) = off) /\
(SM0out ACTIVE (trap (PR reset)) = on) /\
(SM0out ACTIVE (trap (PR launch)) = on) /\
(SM0out ACTIVE (trap (NP status)) = on) /\
(SM0out STBY discard = off) /\
(SM0out ACTIVE discard = on)’

```

[SM0out\_def]

```

⊢ (SM0out STBY (exec (PR reset)) = off) ∧
  (SM0out STBY (exec (PR launch)) = on) ∧
  (SM0out STBY (exec (NP status)) = off) ∧
  (SM0out ACTIVE (exec (PR reset)) = off) ∧
  (SM0out ACTIVE (exec (PR launch)) = on) ∧
  (SM0out ACTIVE (exec (NP status)) = on) ∧
  (SM0out STBY (trap (PR reset)) = off) ∧
  (SM0out STBY (trap (PR launch)) = off) ∧
  (SM0out STBY (trap (NP status)) = off) ∧
  (SM0out ACTIVE (trap (PR reset)) = on) ∧
  (SM0out ACTIVE (trap (PR launch)) = on) ∧
  (SM0out ACTIVE (trap (NP status)) = on) ∧
  (SM0out STBY discard = off) ∧ (SM0out ACTIVE discard = on)

```

### 17.2.4 Input Authentication Function

The following code snippet defines the `inputOK` input authentication function that recognizes Alice and Bob but not Carol.

```

val inputOK_def =
Define
'(inputOK
  (((Name Alice) says
    (prop (SOME (cmd:command)))):(command inst ,staff ,'d,'e)Form) = T) /\
(inputOK
  (((Name Bob) says
    (prop (SOME (cmd:command)))):(command inst ,staff ,'d,'e)Form) = T) /\
(inputOK _ = F)'

```

[inputOK\_def]

```

⊢ (inputOK (Name Alice says prop (SOME cmd)) ⇔ T) ∧
  (inputOK (Name Bob says prop (SOME cmd)) ⇔ T) ∧
  (inputOK TT ⇔ F) ∧ (inputOK FF ⇔ F) ∧
  (inputOK (prop v) ⇔ F) ∧ (inputOK (notf v1) ⇔ F) ∧
  (inputOK (v2 andf v3) ⇔ F) ∧ (inputOK (v4 orf v5) ⇔ F) ∧
  (inputOK (v6 impf v7) ⇔ F) ∧ (inputOK (v8 eqf v9) ⇔ F) ∧
  (inputOK (v10 says TT) ⇔ F) ∧
  (inputOK (v10 says FF) ⇔ F) ∧
  (inputOK (Name Carol says prop (SOME v142)) ⇔ F) ∧
  (inputOK (Name v132 says prop NONE) ⇔ F) ∧
  (inputOK (v133 meet v134 says prop v66) ⇔ F) ∧
  (inputOK (v135 quoting v136 says prop v66) ⇔ F) ∧
  (inputOK (v10 says notf v67) ⇔ F) ∧
  (inputOK (v10 says (v68 andf v69)) ⇔ F) ∧
  (inputOK (v10 says (v70 orf v71)) ⇔ F) ∧
  (inputOK (v10 says (v72 impf v73)) ⇔ F) ∧
  (inputOK (v10 says (v74 eqf v75)) ⇔ F) ∧
  (inputOK (v10 says v76 says v77) ⇔ F) ∧
  (inputOK (v10 says v78 speaks_for v79) ⇔ F) ∧

```



```

(inputOK (v10 says v80 controls v81)  $\iff$  F)  $\wedge$ 
(inputOK (v10 says reps v82 v83 v84)  $\iff$  F)  $\wedge$ 
(inputOK (v10 says v85 domi v86)  $\iff$  F)  $\wedge$ 
(inputOK (v10 says v87 eqi v88)  $\iff$  F)  $\wedge$ 
(inputOK (v10 says v89 doms v90)  $\iff$  F)  $\wedge$ 
(inputOK (v10 says v91 eqs v92)  $\iff$  F)  $\wedge$ 
(inputOK (v10 says v93 eqn v94)  $\iff$  F)  $\wedge$ 
(inputOK (v10 says v95 lte v96)  $\iff$  F)  $\wedge$ 
(inputOK (v10 says v97 lt v98)  $\iff$  F)  $\wedge$ 
(inputOK (v12 speaks_for v13)  $\iff$  F)  $\wedge$ 
(inputOK (v14 controls v15)  $\iff$  F)  $\wedge$ 
(inputOK (reps v16 v17 v18)  $\iff$  F)  $\wedge$ 
(inputOK (v19 domi v20)  $\iff$  F)  $\wedge$ 
(inputOK (v21 eqi v22)  $\iff$  F)  $\wedge$ 
(inputOK (v23 doms v24)  $\iff$  F)  $\wedge$ 
(inputOK (v25 eqs v26)  $\iff$  F)  $\wedge$ 
(inputOK (v27 eqn v28)  $\iff$  F)  $\wedge$ 
(inputOK (v29 lte v30)  $\iff$  F)  $\wedge$  (inputOK (v31 lt v32)  $\iff$  F)

```

### 17.2.5 Certificates Establishing Security Context

The following code snippet defines the `certs` function that establishes the security context for  $SM_0$ .

```

val certs_def =
  Define
  'certs (cmd:command) (npriv:npriv) (privcmd:privcmd) =
  [(Name Alice controls ((prop (SOME (NP npriv))): (command inst, staff, 'd,'e)Form));
   Name Alice controls (prop (SOME (PR privcmd)));
   Name Bob controls prop (SOME (NP npriv));
   ((Name Bob) says (prop (SOME (PR privcmd)))) impf (prop NONE)]

```

[certs\_def]

```

 $\vdash \forall cmd \ npriv \ privcmd.$ 
  certs cmd npriv privcmd =
  [Name Alice controls prop (SOME (NP npriv));
   Name Alice controls prop (SOME (PR privcmd));
   Name Bob controls prop (SOME (NP npriv));
   Name Bob says prop (SOME (PR privcmd)) impf prop NONE]

```

### 17.2.6 Proof that Carol's Commands are Discarded

Given the definition of `inputOK`, which accepts commands only from Alice and Bob, Carol's commands are viewed as unauthenticated and are rejected. This is the theorem *Carol\_discard\_lemma*.

[Carol\_discard\_lemma]

```

 $\vdash$  TR ( $M, O_i, O_s$ ) discard
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
   (Name Carol says prop (SOME cmd)::ins) s outs)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
   (SM0ns s discard) (SM0out s discard::outs))

```

The proof is straightforward and is based on the fact that given the definition of `inputOK`, the application of `inputOK` to `Name Carol says prop (SOME cmd)` returns *false*. This is the theorem *Carol\_rejected\_lemma*.

### Example 17.1

We prove that inputs of the form `Name Carol says prop (SOME cmd)` are rejected by the function `inputOK`. This is a straightforward proof relying only on the definition *inputOK\_def*. The following code snippet prove the theorem.

```
[Carol_rejected_lemma]
⊢ ¬inputOK (Name Carol says prop (SOME cmd))
```

```
val Carol_rejected_lemma =
TAC_PROOF([],
  ``inputOK
    (((Name Carol) says (prop (SOME (cmd:command)))):(command inst,staff,'d','e')Form)``),
PROVE_TAC[inputOK_def])
```

Executing the above produces the following transcript and theorem.

```
- val Carol_rejected_lemma =
TAC_PROOF([],
  ``inputOK
    (((Name Carol) says (prop (SOME (cmd:command)))):(command inst,staff,'d','e')Form)``),
PROVE_TAC[inputOK_def]);
Meson search level: ..
> val Carol_rejected_lemma =
  |- ~inputOK (Name Carol says prop (SOME cmd))
  : thm
```

◇

### Example 17.2

Using *Carol\_rejected\_lemma*, it is straightforward to prove that a starting configuration with a command originating from Carol, i.e.,

```
CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
(Name Carol says prop (SOME cmd)::ins) s outs,
```

is discarded, resulting in the following configuration

```
CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
(SM0ns s discard) (SM0out s discard::outs).
```

The code snippet below produces the theorem and the following transcript.

```
[Carol_discard_lemma]
⊢ TR (M, Oi, Os) discard
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
    (Name Carol says prop (SOME cmd)::ins) s outs)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
    (SM0ns s discard) (SM0out s discard::outs))
```

```

val Carol_discard_lemma =
TAC.PROOF([],
  ``TR ((M:(command inst,'b,staff,'d,'e)Kripke),Oi,Os) discard
    (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
      (((Name Carol) says (prop (SOME (cmd:command))))::ins)
      s (outs:output list))
    (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
      (SM0ns s discard) ((SM0out s discard)::outs))``,
  PROVE_TAC[Carol_rejected_lemma, TR_discard_cmd_rule])

```

2

```

- val Alice_privcmd_lemma =
TAC_PROOF([],
  ``CFGInterpret ((M:(command inst,'b,staff,'d,'e)Kripke),Oi,Os)
    (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
      ((Name Alice) says (prop (SOME (PR (privcmd:privcmd))))::ins)
      s (outs:output list)) ==>
    (M,Oi,Os) sat (prop (SOME (PR privcmd))))``,
  REWRITE_TAC[CFGInterpret_def, certs_def, SM0StateInterp_def, satList_CONS,
    satList_nil, sat_TT] THEN
  PROVE_TAC[Controls]);
Meson search level: ....
> val Alice_privcmd_lemma =
  |- CFGInterpret (M,Oi,Os)
    (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
      (Name Alice says prop (SOME (PR privcmd))::ins) s outs) ==>
    (M,Oi,Os) sat prop (SOME (PR privcmd))
  : thm

```

◇

## 17.2.7 Proof that Alice's Commands are Completely Mediated

Our objective is to show that Alice's commands are completely mediated. This means that (1) if her commands are executed, then they necessarily are authenticated and authorized, and (2) if her commands are authenticated and authorized, then they are executed.

1. The theorem corresponding to the first condition is *Alice\_privcmd\_verified\_thm*, shown below.

[\[Alice\\_privcmd\\_verified\\_thm\]](#)

$$\begin{aligned}
 &\vdash \text{TR } (M, Oi, Os) \text{ (exec (PR privcmd))} \\
 &\quad (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)} \\
 &\quad \quad (\text{Name Alice says prop (SOME (PR privcmd))::ins) s outs}) \\
 &\quad (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins} \\
 &\quad \quad (\text{NS s (exec (PR privcmd))}) \\
 &\quad \quad (\text{Out s (exec (PR privcmd))::outs})) \Rightarrow \\
 & (M, Oi, Os) \text{ sat prop (SOME (PR privcmd))}
 \end{aligned}$$

This theorem states that if an `exec (PR privcmd)` transition occurs due to a command

`Name Alice says prop (SOME (PR privcmd))`,

then executing is justified in the access-control logic, as given by

$$(M, Oi, Os) \text{ sat prop } (\text{SOME } (\text{PR } \text{privcmd})) .$$

2. The theorem corresponding to the second condition is *Alice\_justified\_privcmd\_exec\_thm*, shown below.

```
[Alice_justified_privcmd_exec_thm]
⊢ inputOK (Name Alice says prop (SOME (PR privcmd))) ∧
CFGInterpret (M, Oi, Os)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
   (Name Alice says prop (SOME (PR privcmd)) :: ins) s
   outs) ⇒
TR (M, Oi, Os) (exec (PR privcmd))
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
   (Name Alice says prop (SOME (PR privcmd)) :: ins) s outs)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
   (NS s (exec (PR privcmd))))
  (Out s (exec (PR privcmd)) :: outs)
```

This theorem states that if the input passes the integrity check and *CFGInterpret* provides the security interpretation of the configuration in the access-control logic, i.e.,

```
inputOK (Name Alice says prop (SOME (PR privcmd))) ∧
CFGInterpret (M, Oi, Os)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
   (Name Alice says prop (SOME (PR privcmd)) :: ins) s outs),
```

then executing the command, as shown by the *exec (PR privcmd)* transition, is justified, i.e.,

```
TR (M, Oi, Os) (exec (PR privcmd))
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
   (Name Alice says prop (SOME (PR privcmd)) :: ins) s outs)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
   (NS s (exec (PR privcmd))))
  (Out s (exec (PR privcmd)) :: outs)
```

To prove the two theorems above, we first prove the theorem *Alice\_exec\_privcmd\_justified\_thm*, which is shown below. Once the theorem is proved, the two theorems above are easily proved.

```
[Alice_exec_privcmd_justified_thm]
⊢ TR (M, Oi, Os) (exec (PR privcmd))
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
   (Name Alice says prop (SOME (PR privcmd)) :: ins) s outs)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
```

$$\begin{aligned}
& (NS\ s\ (\text{exec}\ (\text{PR}\ \text{privcmd}))) \\
& (\text{Out}\ s\ (\text{exec}\ (\text{PR}\ \text{privcmd})) :: \text{outs})) \iff \\
& \text{inputOK}\ (\text{Name Alice says prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd}))) \wedge \\
& \text{CFGInterpret}\ (M, Oi, Os) \\
& (\text{CFG inputOK SM0StateInterp}\ (\text{certs}\ \text{cmd npriv privcmd}) \\
& (\text{Name Alice says prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd})) :: \text{ins})\ s \\
& \text{outs}) \wedge (M, Oi, Os)\ \text{sat prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd}))
\end{aligned}$$

The next three examples take us through the proofs.

### Example 17.3

Proving *Alice\_exec\_privcmd\_justified\_thm* has two components.

1. Proving that the interpretation of the starting configuration justifies executing Alice's command, i.e.,

$$\begin{aligned}
& [\text{Alice\_privcmd\_lemma}] \\
& \vdash \text{CFGInterpret}\ (M, Oi, Os) \\
& (\text{CFG inputOK SM0StateInterp}\ (\text{certs}\ \text{cmd npriv privcmd}) \\
& (\text{Name Alice says prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd})) :: \text{ins})\ s \\
& \text{outs}) \Rightarrow \\
& (M, Oi, Os)\ \text{sat prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd})),
\end{aligned}$$

and

2. Specializing the variables of *TR\_exec\_cmd\_rule* to match the parameters in the starting configuration, to get

$$\begin{aligned}
& (\forall M\ Oi\ Os. \\
& \text{CFGInterpret}\ (M, Oi, Os) \\
& (\text{CFG inputOK SM0StateInterp}\ (\text{certs}\ \text{cmd npriv privcmd}) \\
& (\text{Name Alice says prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd})) :: \text{ins})\ s \\
& \text{outs}) \Rightarrow \\
& (M, Oi, Os)\ \text{sat prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd})) \Rightarrow \\
& \forall NS\ \text{Out}\ M\ Oi\ Os. \\
& \text{TR}\ (M, Oi, Os)\ (\text{exec}\ (\text{PR}\ \text{privcmd})) \\
& (\text{CFG inputOK SM0StateInterp}\ (\text{certs}\ \text{cmd npriv privcmd}) \\
& (\text{Name Alice says prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd})) :: \text{ins})\ s\ \text{outs}) \\
& (\text{CFG inputOK SM0StateInterp}\ (\text{certs}\ \text{cmd npriv privcmd})\ \text{ins} \\
& (NS\ s\ (\text{exec}\ (\text{PR}\ \text{privcmd}))) \\
& (\text{Out}\ s\ (\text{exec}\ (\text{PR}\ \text{privcmd})) :: \text{outs})) \iff \\
& \text{inputOK}\ (\text{Name Alice says prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd}))) \wedge \\
& \text{CFGInterpret}\ (M, Oi, Os) \\
& (\text{CFG inputOK SM0StateInterp}\ (\text{certs}\ \text{cmd npriv privcmd}) \\
& (\text{Name Alice says prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd})) :: \text{ins})\ s \\
& \text{outs}) \wedge (M, Oi, Os)\ \text{sat prop}\ (\text{SOME}\ (\text{PR}\ \text{privcmd}))
\end{aligned}$$

- A. The proof of *Alice\_privcmd\_lemma* is done by rewriting using the definitions of *CFGInterpret*, *certs*, and *SM0StateInterp*, and the properties *satList\_CONS*, *satList\_nil*, and *sat\_TT*, where *sat\_TT* is in *aclrulesTheory*. After the goal is rewritten, the proof is completed by *PROVE\_TAC* with the *Controls* theorem, which is part of *aclDrulesTheory*.

```
[sat_TT] ⊢ (M, Oi, Os) sat TT

[Controls] ⊢ (M, Oi, Os) sat P says f ⇒
(M, Oi, Os) sat P controls f ⇒
(M, Oi, Os) sat f
```

The proof code is as follows.

```
val Alice_privcmd_lemma =
TAC.PROOF(
  ([], ``CFGInterpret
    ((M:(command inst,'b,staff,'d,'e)Kripke),Oi,Os)
    (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
      (((Name Alice) says (prop (SOME (PR (privcmd:privcmd))))::ins) s
      (outs:output list)) ==>
    ((M,Oi,Os) sat (prop (SOME(PR privcmd)))) ``),
  REWRITE_TAC[CFGInterpret_def, certs_def, SM0StateInterp_def, satList_CONS,
    satList_nil, sat_TT] THEN
  PROVE_TAC[Controls])

val _ = save_thm("Alice_privcmd_lemma", Alice_privcmd_lemma)
```

Executing the above produces proves the theorem, as shown below.

```
- val Alice_privcmd_lemma =
TAC_PROOF([],
  ``CFGInterpret ((M:(command inst,'b,staff,'d,'e)Kripke),Oi,Os)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
    (((Name Alice) says (prop (SOME (PR (privcmd:privcmd))))::ins)
    s (outs:output list)) ==>
    ((M,Oi,Os) sat (prop (SOME(PR privcmd)))) ``),
  REWRITE_TAC[CFGInterpret_def, certs_def, SM0StateInterp_def, satList_CONS,
    satList_nil, sat_TT] THEN
  PROVE_TAC[Controls]);
Meson search level: ....
> val Alice_privcmd_lemma =
  |- CFGInterpret (M,Oi,Os)
    (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
      (Name Alice says prop (SOME (PR privcmd))))::ins) s outs) ==>
    (M,Oi,Os) sat prop (SOME (PR privcmd))
  : thm
```

3

- B. The proof of *Alice\_exec\_privcmd\_justified\_thm* is done in two steps. The first step specializes the parameters of *TR\_exec\_cmd\_rule* to match the parameters of the starting configuration. This is done using the forward inference rule *ISPECL* by the code snippet below.

```
val th1 =
ISPECL
  [ ``inputOK:(command inst, staff,'d,'e)Form -> bool``,
    ``(certs cmd npriv privcmd):(command inst, staff,'d,'e)Form list``,
    ``SM0StateInterp:state ->(command inst, staff,'d,'e)Form``,
```

```

    ``Name Alice``, ``PR privcmd``, ``ins:(command inst, staff, 'd, 'e)Form list``,
    ``s:state``, ``outs:output list``]
  TR_exec_cmd_rule

```

The above produces the specialized theorem shown below.

```

- val th1 =
  ISPECL
  [ ``inputOK:(command inst, staff, 'd, 'e)Form -> bool``,
    ``(certs cmd npriv privcmd):(command inst, staff, 'd, 'e)Form list``,
    ``SM0StateInterp:state->(command inst, staff, 'd, 'e)Form``,
    ``Name Alice``, ``PR privcmd``, ``ins:(command inst, staff, 'd, 'e)Form list``,
    ``s:state``, ``outs:output list``]
  TR_exec_cmd_rule;
> val th1 =
  |- (!M Oi Os.
    CFGInterpret (M,Oi,Os)
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
        (Name Alice says prop (SOME (PR privcmd))::ins) s outs) ==>
      (M,Oi,Os) sat prop (SOME (PR privcmd))) ==>
    !NS Out M Oi Os.
    TR (M,Oi,Os) (exec (PR privcmd))
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
        (Name Alice says prop (SOME (PR privcmd))::ins) s outs)
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
        (NS s (exec (PR privcmd)))
        (Out s (exec (PR privcmd))::outs)) <=>
    inputOK (Name Alice says prop (SOME (PR privcmd))) /\
    CFGInterpret (M,Oi,Os)
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
        (Name Alice says prop (SOME (PR privcmd))::ins) s outs) /\
      (M,Oi,Os) sat prop (SOME (PR privcmd))
  : thm

```

- C. Using *th1* and *Alice\_privcmd\_lemma*, which we just proved, we prove easily the transition equivalence property we want stating that the transition executing Alice's command occurs if and only if her command is authenticated, authorized within the context of *CFGInterpret*, and is justified within the access-control logic.

The following code snippet finishes the proof as described above.

```

TAC.PROOF([],
  ``(NS :state -> command trType -> state)
    (Out :state -> command trType -> output)
    (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
    (Os : 'e po).
  TR (M,Oi,Os) (exec (PR (privcmd :privcmd)))
    (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs (cmd :command) (npriv :npriv) privcmd :
        (command inst, staff, 'd, 'e) Form list)
      (Name Alice says
        (prop (SOME (PR privcmd) :command inst) :
          (command inst, staff, 'd, 'e) Form)::
          (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
      (outs :output list))
    (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs cmd npriv privcmd :
        (command inst, staff, 'd, 'e) Form list) ins
      (NS s (exec (PR privcmd)))
      (Out s (exec (PR privcmd))::outs)) <=>

```

```

inputOK
  (Name Alice says
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M,Oi,Os)
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::ins) s outs) /\
(M,Oi,Os) sat
  (prop (SOME (PR privcmd) :command inst) :
    (command inst, staff, 'd, 'e) Form)='',
PROVE_TAC[th1, Alice_privcmd_lemma])

```

Executing the above code snippet accomplishes the proof, as shown below.



```

- TAC_PROOF([[],
  ``!(NS :state -> command trType -> state)
    (Out :state -> command trType -> output)
    (M : (command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
    (Os : 'e po).
  TR (M,Oi,Os) (exec (PR (privcmd :privcmd)))
    (CFG (inputOK : (command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs (cmd :command) (npriv :npriv) privcmd :
        (command inst, staff, 'd, 'e) Form list)
      (Name Alice says
        (prop (SOME (PR privcmd) :command inst) :
          (command inst, staff, 'd, 'e) Form)::
          (ins : (command inst, staff, 'd, 'e) Form list)) (s :state)
        (outs :output list))
      (CFG (inputOK : (command inst, staff, 'd, 'e) Form -> bool)
        (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
        (certs cmd npriv privcmd :
          (command inst, staff, 'd, 'e) Form list) ins
        (NS s (exec (PR privcmd)))
        (Out s (exec (PR privcmd))::outs)) <=>
      inputOK
      (Name Alice says
        (prop (SOME (PR privcmd) :command inst) :
          (command inst, staff, 'd, 'e) Form)) /\
      CFGInterpret (M,Oi,Os)
      (CFG (inputOK : (command inst, staff, 'd, 'e) Form -> bool)
        (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
        (certs cmd npriv privcmd :
          (command inst, staff, 'd, 'e) Form list)
        (Name Alice says
          (prop (SOME (PR privcmd) :command inst) :
            (command inst, staff, 'd, 'e) Form)::ins) s outs) /\
      (M,Oi,Os) sat
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form))``,
  PROVE_TAC[th1,Alice_privcmd_lemma]);
Meson search level: .....
> val it =
  |- !NS Out M Oi Os.
    TR (M,Oi,Os) (exec (PR privcmd))
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
        (Name Alice says prop (SOME (PR privcmd))::ins) s outs)
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
        (NS s (exec (PR privcmd)))
        (Out s (exec (PR privcmd))::outs)) <=>
      inputOK (Name Alice says prop (SOME (PR privcmd))) /\
      CFGInterpret (M,Oi,Os)
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
        (Name Alice says prop (SOME (PR privcmd))::ins) s outs) /\
      (M,Oi,Os) sat prop (SOME (PR privcmd))
: thm

```

As always, we combine the steps above into one function that does the whole proof. The code for this is as follows.

```

val Alice_exec_privcmd_justified_thm =
let
  val th1 =
  ISPECL
  [ ``inputOK:(command inst, staff,'d,'e)Form -> bool``,
    ``(certs cmd npriv privcmd):(command inst, staff,'d,'e)Form list``,
    ``SM0StateInterp:state ->(command inst, staff,'d,'e)Form``,
    ``Name Alice``, ``PR privcmd``, ``ins:(command inst,staff,'d,'e)Form list``,
    ``s:state``, ``outs:output list`` ]
  TR_exec_cmd_rule

```

```

in
  TAC_PROOF([
    “!(NS :state -> command trType -> state)
      (Out :state -> command trType -> output)
      (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi :’d po)
      (Os :’e po).
    TR (M,Oi,Os) (exec (PR (privcmd :privcmd)))
      (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
        (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
        (certs (cmd :command) (npriv :npriv) privcmd :
          (command inst, staff, 'd, 'e) Form list)
        (Name Alice says
          (prop (SOME (PR privcmd) :command inst) :
            (command inst, staff, 'd, 'e) Form)::
            (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
          (outs :output list))
        (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
          (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
          (certs cmd npriv privcmd :
            (command inst, staff, 'd, 'e) Form list) ins
          (NS s (exec (PR privcmd)))
          (Out s (exec (PR privcmd))::outs)) <=>
      inputOK
        (Name Alice says
          (prop (SOME (PR privcmd) :command inst) :
            (command inst, staff, 'd, 'e) Form)) /\
      CFGInterpret (M,Oi,Os)
        (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
          (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
          (certs cmd npriv privcmd :
            (command inst, staff, 'd, 'e) Form list)
          (Name Alice says
            (prop (SOME (PR privcmd) :command inst) :
              (command inst, staff, 'd, 'e) Form)::ins) s outs) /\
        (M,Oi,Os) sat
        (prop (SOME (PR privcmd) :command inst) :
          (command inst, staff, 'd, 'e) Form)“),
    PROVE_TAC[th1, Alice_privcmd_lemma])
  end

val _ = save_thm("Alice_exec_privcmd_justified_thm", Alice_exec_privcmd_justified_thm)

```

Executing the above produces the transcript below.

```

- val Alice_exec_privcmd_justified_thm =
let
  val th1 =
    ISPECL
    [``inputOK:(command inst, staff,'d,'e)Form -> bool``,
      ``(certs cmd npriv privcmd):(command inst, staff,'d,'e)Form list``,
      ``SM0StateInterp:state->(command inst, staff,'d,'e)Form``,
      ``Name Alice``, ``PR privcmd``, ``ins:(command inst,staff,'d,'e)Form list``,
      ``s:state``, ``outs:output list``]
    TR_exec_cmd_rule
in
  TAC_PROOF([[],
    ``(NS :state -> command trType -> state)
      (Out :state -> command trType -> output)
      (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi :'d po)
      (Os :'e po).
      TR (M,Oi,Os) (exec (PR (privcmd :privcmd)))
      (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
        (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
        (certs (cmd :command) (npriv :npriv) privcmd :
          (command inst, staff, 'd, 'e) Form list)
        (Name Alice says
          (prop (SOME (PR privcmd) :command inst) :
            (command inst, staff, 'd, 'e) Form)::
            (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
          (outs :output list))
        (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
          (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
          (certs cmd npriv privcmd :
            (command inst, staff, 'd, 'e) Form list) ins
          (NS s (exec (PR privcmd)))
          (Out s (exec (PR privcmd))::outs)) <=>
        inputOK
          (Name Alice says
            (prop (SOME (PR privcmd) :command inst) :
              (command inst, staff, 'd, 'e) Form)) /\
          CFGInterpret (M,Oi,Os)
            (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
              (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
              (certs cmd npriv privcmd :
                (command inst, staff, 'd, 'e) Form list)
              (Name Alice says
                (prop (SOME (PR privcmd) :command inst) :
                  (command inst, staff, 'd, 'e) Form)::ins) s outs) /\
            (M,Oi,Os) sat
            (prop (SOME (PR privcmd) :command inst) :
              (command inst, staff, 'd, 'e) Form))``,
      PROVE_TAC[th1,Alice_privcmd_lemma])
  end;
  Meson search level: .....
  > val Alice_exec_privcmd_justified_thm =
    |- !NS Out M Oi Os.
      TR (M,Oi,Os) (exec (PR privcmd))
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
        (Name Alice says prop (SOME (PR privcmd))::ins) s outs)
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
        (NS s (exec (PR privcmd)))
        (Out s (exec (PR privcmd))::outs)) <=>
      inputOK (Name Alice says prop (SOME (PR privcmd))) /\
      CFGInterpret (M,Oi,Os)
        (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
          (Name Alice says prop (SOME (PR privcmd))::ins) s outs) /\
        (M,Oi,Os) sat prop (SOME (PR privcmd))
    : thm

```



**Example 17.4**

The proof of *Alice\_privcmd\_verified\_thm* follows directly from the theorem *Alice\_exec\_privcmd\_justified\_thm* using `PROVE_TAC`. The code snippet below proves *Alice\_privcmd\_verified\_thm*.

```
[Alice_privcmd_verified_thm]
⊢ TR (M, Oi, Os) (exec (PR privcmd))
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
    (Name Alice says prop (SOME (PR privcmd)) :: ins) s outs)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
    (NS s (exec (PR privcmd))))
  (Out s (exec (PR privcmd)) :: outs)) ⇒
(M, Oi, Os) sat prop (SOME (PR privcmd))
```

```
val Alice_privcmd_verified_thm =
TAC.PROOF([], '!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi :'d po)
  (Os :'e po).
  TR (M.Oi.Os) (exec (PR (privcmd :privcmd)))
    (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs (cmd :command) (npriv :npriv) privcmd :
        (command inst, staff, 'd, 'e) Form list)
      (Name Alice says
        (prop (SOME (PR privcmd) :command inst) :
          (command inst, staff, 'd, 'e) Form)::
          (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
        (outs :output list))
      (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
        (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
        (certs cmd npriv privcmd :
          (command inst, staff, 'd, 'e) Form list) ins
        (NS s (exec (PR privcmd)))
        (Out s (exec (PR privcmd)) :: outs)) ==>
    (M, Oi, Os) sat
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form) ' '),
PROVE_TAC[ Alice_exec_privcmd_justified_thm ])

val _ = save_thm("Alice_privcmd_verified_thm", Alice_privcmd_verified_thm)
```

The above code produces the following transcript.

```

- val Alice_privcmd_verified_thm =
TAC_PROOF([[], ``!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M : (command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po).
  TR (M, Oi, Os) (exec (PR (privcmd :privcmd)))
    (CFG (inputOK : (command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs (cmd :command) (npriv :npriv) privcmd :
        (command inst, staff, 'd, 'e) Form list)
      (Name Alice says
        (prop (SOME (PR privcmd) :command inst) :
          (command inst, staff, 'd, 'e) Form)::
          (ins : (command inst, staff, 'd, 'e) Form list)) (s :state)
        (outs :output list))
      (CFG (inputOK : (command inst, staff, 'd, 'e) Form -> bool)
        (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
        (certs cmd npriv privcmd :
          (command inst, staff, 'd, 'e) Form list) ins
        (NS s (exec (PR privcmd)))
        (Out s (exec (PR privcmd))::outs)) ==>
      (M, Oi, Os) sat
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)`),
  PROVE_TAC[Alice_exec_privcmd_justified_thm];
Meson search level: ...
> val Alice_privcmd_verified_thm =
  |- !NS Out M Oi Os.
    TR (M, Oi, Os) (exec (PR privcmd))
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
        (Name Alice says prop (SOME (PR privcmd))::ins) s outs)
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
        (NS s (exec (PR privcmd)))
        (Out s (exec (PR privcmd))::outs)) ==>
      (M, Oi, Os) sat prop (SOME (PR privcmd))
: thm

```

◇

### Example 17.5

The theorem *Alice\_justified\_privcmd\_exec\_thm* states that if Alice's commands are authenticated and authorized, then they are executed. The theorem follows directly from the definition of *inputOK*, and the theorems *Alice\_privcmd\_lemma* and *Alice\_justified\_privcmd\_exec\_thm*.

```

[Alice_justified_privcmd_exec_thm]
⊢ inputOK (Name Alice says prop (SOME (PR privcmd))) ∧
CFGInterpret (M, Oi, Os)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
    (Name Alice says prop (SOME (PR privcmd))::ins) s
    outs) ⇒
TR (M, Oi, Os) (exec (PR privcmd))
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
    (Name Alice says prop (SOME (PR privcmd))::ins) s outs)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
    (NS s (exec (PR privcmd)))
    (Out s (exec (PR privcmd))::outs))

```

```

val Alice_justified_privcmd_exec_thm =
TAC.PROOF([[], '!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi :'d po)
  (Os :'e po) cmd npriv privcmd ins s outs.
inputOK
  (Name Alice says
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form)) /\
  CFGInterpret (M,Oi,Os)
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::ins) s outs) ==>
  TR (M,Oi,Os) (exec (PR (privcmd :privcmd)))
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::
        (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
      (outs :output list))
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list) ins
    (NS s (exec (PR privcmd)))
    (Out s (exec (PR privcmd))::outs)) ' '),
PROVE_TAC[ Alice_exec_privcmd_justified_thm, inputOK_def, Alice_privcmd_lemma ]

val _ = save_thm("Alice_justified_privcmd_exec_thm", Alice_justified_privcmd_exec_thm)

```

The above code produces the following transcript.

```

- val Alice_justified_privcmd_exec_thm =
TAC_PROOF([[], ``!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M : (command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po) cmd npriv privcmd ins s outs.

inputOK
  (Name Alice says
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M,Oi,Os)
  (CFG (inputOK : (command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::ins) s outs) ==>
TR (M,Oi,Os) (exec (PR (privcmd :privcmd)))
  (CFG (inputOK : (command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::
        (ins : (command inst, staff, 'd, 'e) Form list)) (s :state)
      (outs :output list))
  (CFG (inputOK : (command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list) ins
    (NS s (exec (PR privcmd)))
    (Out s (exec (PR privcmd))::outs)) ``),
PROVE_TAC[Alice_exec_privcmd_justified_thm,inputOK_def,Alice_privcmd_lemma];
Meson search level: .....
> val Alice_justified_privcmd_exec_thm =
  |- !NS Out M Oi Os cmd npriv privcmd ins s outs.
    inputOK (Name Alice says prop (SOME (PR privcmd))) /\
    CFGInterpret (M,Oi,Os)
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
        (Name Alice says prop (SOME (PR privcmd))::ins) s outs) ==>
    TR (M,Oi,Os) (exec (PR privcmd))
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
        (Name Alice says prop (SOME (PR privcmd))::ins) s outs)
      (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
        (NS s (exec (PR privcmd))) (Out s (exec (PR privcmd))::outs))

: thm

```

◇

## 17.3 Exercises

**Exercise 17.3.1** *Using inputOK and certs to authenticate and authorize commands, prove the following theorems that justify Alice's request to execute a non-privileged command will be executed.*

A. [\[Alice\\_npriv\\_lemma\]](#)

$$\begin{aligned}
 &\vdash \text{CFGInterpret } (M, Oi, Os) \\
 &(\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)} \\
 &\quad (\text{Name Alice says prop (SOME (NP npriv))::ins) s outs) \Rightarrow
 \end{aligned}$$

$(M, Oi, Os) \text{ sat prop (SOME (NP npriv))}$

B. *[Alice\_exec\_npriv\_justified\_thm]*

$\vdash \text{TR } (M, Oi, Os) \text{ (exec (NP npriv))}$   
 $(\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)}$   
 $\quad (\text{Name Alice says prop (SOME (NP npriv)) :: ins) s outs})$   
 $(\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins}$   
 $\quad (\text{NS s (exec (NP npriv))})$   
 $\quad (\text{Out s (exec (NP npriv)) :: outs})) \iff$   
 $\text{inputOK (Name Alice says prop (SOME (NP npriv)))} \wedge$   
 $\text{CFGInterpret } (M, Oi, Os)$   
 $\quad (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)}$   
 $\quad \quad (\text{Name Alice says prop (SOME (NP npriv)) :: ins) s outs}) \wedge$   
 $(M, Oi, Os) \text{ sat prop (SOME (NP npriv))}$

C. *[Alice\_npriv\_verified\_thm]*

$\vdash \text{TR } (M, Oi, Os) \text{ (exec (NP npriv))}$   
 $(\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)}$   
 $\quad (\text{Name Alice says prop (SOME (NP npriv)) :: ins) s outs})$   
 $(\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins}$   
 $\quad (\text{NS s (exec (NP npriv))})$   
 $\quad (\text{Out s (exec (NP npriv)) :: outs})) \Rightarrow$   
 $(M, Oi, Os) \text{ sat prop (SOME (NP npriv))}$

D. *[Alice\_justified\_npriv\_exec\_thm]*

$\vdash \text{inputOK (Name Alice says prop (SOME (NP npriv)))} \wedge$   
 $\text{CFGInterpret } (M, Oi, Os)$   
 $\quad (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)}$   
 $\quad \quad (\text{Name Alice says prop (SOME (NP npriv)) :: ins) s outs}) \Rightarrow$   
 $\text{TR } (M, Oi, Os) \text{ (exec (NP npriv))}$   
 $\quad (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)}$   
 $\quad \quad (\text{Name Alice says prop (SOME (NP npriv)) :: ins) s outs})$   
 $(\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins}$   
 $\quad (\text{NS s (exec (NP npriv))})$   
 $\quad (\text{Out s (exec (NP npriv)) :: outs}))$

**Exercise 17.3.2** Using inputOK and certs to authenticate and authorize commands, prove the following theorems that justify trapping Bob's request to execute a privileged command. **Hint:** the following theorems will help you:

- Modus\_Ponens inference rule of aclrulesTheory.



- TR\_trap\_cmd\_rule of ssm1Theory.

A. [\[Bob\\_privcmd\\_trap\\_lemma\]](#)

$$\begin{aligned} & \vdash \text{CFGInterpret } (M, Oi, Os) \\ & (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)} \\ & \quad (\text{Name Bob says prop (SOME (PR privcmd)) :: ins) s outs}) \Rightarrow \\ & (M, Oi, Os) \text{ sat prop NONE} \end{aligned}$$

B. [\[Bob\\_trap\\_privcmd\\_justified\\_thm\]](#)

$$\begin{aligned} & \vdash \text{TR } (M, Oi, Os) (\text{trap (PR privcmd)}) \\ & (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)} \\ & \quad (\text{Name Bob says prop (SOME (PR privcmd)) :: ins) s outs}) \\ & (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins} \\ & \quad (\text{NS s (trap (PR privcmd))}) \\ & \quad (\text{Out s (trap (PR privcmd)) :: outs})) \iff \\ & \text{inputOK (Name Bob says prop (SOME (PR privcmd)))} \wedge \\ & \text{CFGInterpret } (M, Oi, Os) \\ & (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)} \\ & \quad (\text{Name Bob says prop (SOME (PR privcmd)) :: ins) s outs}) \wedge \\ & (M, Oi, Os) \text{ sat prop NONE} \end{aligned}$$

C. [\[Bob\\_privcmd\\_trapped\\_thm\]](#)

$$\begin{aligned} & \vdash \text{TR } (M, Oi, Os) (\text{trap (PR privcmd)}) \\ & (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)} \\ & \quad (\text{Name Bob says prop (SOME (PR privcmd)) :: ins) s outs}) \\ & (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins} \\ & \quad (\text{NS s (trap (PR privcmd))}) \\ & \quad (\text{Out s (trap (PR privcmd)) :: outs})) \Rightarrow \\ & (M, Oi, Os) \text{ sat prop NONE} \end{aligned}$$

D. [\[Bob\\_justified\\_privcmd\\_trap\\_thm\]](#)

$$\begin{aligned} & \vdash \text{inputOK (Name Bob says prop (SOME (PR privcmd)))} \wedge \\ & \text{CFGInterpret } (M, Oi, Os) \\ & (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)} \\ & \quad (\text{Name Bob says prop (SOME (PR privcmd)) :: ins) s outs}) \Rightarrow \\ & \text{TR } (M, Oi, Os) (\text{trap (PR privcmd)}) \\ & (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd)} \\ & \quad (\text{Name Bob says prop (SOME (PR privcmd)) :: ins) s outs}) \\ & (\text{CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins} \\ & \quad (\text{NS s (trap (PR privcmd))}) \\ & \quad (\text{Out s (trap (PR privcmd)) :: outs})) \end{aligned}$$

**Exercise 17.3.3**

- A. Devise two new definitions **inputOK2** and **certs2**, within `SM0Script.sml`, that authenticate **only** Carol and authorize her to execute non-privileged instructions. The new definitions should reflect the fact that if Carol attempts to execute a privileged command, her request is trapped. Any inputs by Alice or Bob are rejected by **inputOK2**.
- B. Using **inputOK2** and **certs2**, prove the following theorems that justify executing Carol's request to execute a non-privileged command.

(1) [\[Carol\\_npriv\\_lemma\]](#)

$$\begin{aligned} & \vdash \text{CFGInterpret } (M, Oi, Os) \\ & (\text{CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)} \\ & \quad (\text{Name Carol says prop (SOME (NP npriv)) :: ins} s outs) \Rightarrow \\ & (M, Oi, Os) \text{ sat prop (SOME (NP npriv))} \end{aligned}$$

(2) [\[Carol\\_exec\\_npriv\\_justified\\_thm\]](#)

$$\begin{aligned} & \vdash \text{TR } (M, Oi, Os) (\text{exec (NP npriv)}) \\ & (\text{CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)} \\ & \quad (\text{Name Carol says prop (SOME (NP npriv)) :: ins} s outs) \\ & (\text{CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)} \\ & \quad \text{ins (NS s (exec (NP npriv)))} \\ & \quad (\text{Out s (exec (NP npriv)) :: outs})) \iff \\ & \text{inputOK2 (Name Carol says prop (SOME (NP npriv)))} \wedge \\ & \text{CFGInterpret } (M, Oi, Os) \\ & (\text{CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)} \\ & \quad (\text{Name Carol says prop (SOME (NP npriv)) :: ins} s outs) \wedge \\ & (M, Oi, Os) \text{ sat prop (SOME (NP npriv))} \end{aligned}$$

(3) [\[Carol\\_npriv\\_verified\\_thm\]](#)

$$\begin{aligned} & \vdash \text{TR } (M, Oi, Os) (\text{exec (NP npriv)}) \\ & (\text{CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)} \\ & \quad (\text{Name Carol says prop (SOME (NP npriv)) :: ins} s outs) \\ & (\text{CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)} \\ & \quad \text{ins (NS s (exec (NP npriv)))} \\ & \quad (\text{Out s (exec (NP npriv)) :: outs})) \Rightarrow \\ & (M, Oi, Os) \text{ sat prop (SOME (NP npriv))} \end{aligned}$$

(4) [\[Carol\\_justified\\_npriv\\_exec\\_thm\]](#)

$$\begin{aligned} & \vdash \text{inputOK2 (Name Carol says prop (SOME (NP npriv)))} \wedge \\ & \text{CFGInterpret } (M, Oi, Os) \\ & (\text{CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)} \\ & \quad (\text{Name Carol says prop (SOME (NP npriv)) :: ins} s outs) \Rightarrow \\ & \text{TR } (M, Oi, Os) (\text{exec (NP npriv)}) \\ & (\text{CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)} \end{aligned}$$

$$\begin{aligned}
& (\text{Name Carol says prop (SOME (NP } npriv)) :: ins) \ s \ outs) \\
& (\text{CFG inputOK2 SM0StateInterp (certs2 cmd } npriv \text{ privcmd)} \\
& \quad \text{ins (NS } s \text{ (exec (NP } npriv))} \\
& \quad \text{(Out } s \text{ (exec (NP } npriv)) :: outs)})
\end{aligned}$$

C. Using inputOK2 and certs2, prove the following theorems that justify trapping Carol's request to execute a privileged command.

(1) [\[Carol\\_privcmd\\_trap\\_lemma\]](#)

$$\begin{aligned}
& \vdash \text{CFGInterpret } (M, Oi, Os) \\
& (\text{CFG inputOK2 SM0StateInterp (certs2 cmd } npriv \text{ privcmd)} \\
& \quad (\text{Name Carol says prop (SOME (PR } privcmd)) :: ins) \ s \\
& \quad \text{outs}) \Rightarrow \\
& (M, Oi, Os) \text{ sat prop NONE}
\end{aligned}$$

(2) [\[Carol\\_trap\\_privcmd\\_justified\\_thm\]](#)

$$\begin{aligned}
& \vdash \text{TR } (M, Oi, Os) \text{ (trap (PR } privcmd)) \\
& (\text{CFG inputOK2 SM0StateInterp (certs2 cmd } npriv \text{ privcmd)} \\
& \quad (\text{Name Carol says prop (SOME (PR } privcmd)) :: ins) \ s \ outs) \\
& (\text{CFG inputOK2 SM0StateInterp (certs2 cmd } npriv \text{ privcmd)} \\
& \quad \text{ins (NS } s \text{ (trap (PR } privcmd))} \\
& \quad \text{(Out } s \text{ (trap (PR } privcmd)) :: outs)}) \iff \\
& \text{inputOK2 (Name Carol says prop (SOME (PR } privcmd)))} \wedge \\
& \text{CFGInterpret } (M, Oi, Os) \\
& (\text{CFG inputOK2 SM0StateInterp (certs2 cmd } npriv \text{ privcmd)} \\
& \quad (\text{Name Carol says prop (SOME (PR } privcmd)) :: ins) \ s \\
& \quad \text{outs}) \wedge (M, Oi, Os) \text{ sat prop NONE}
\end{aligned}$$

(3) [\[Carol\\_privcmd\\_trapped\\_thm\]](#)

$$\begin{aligned}
& \vdash \text{TR } (M, Oi, Os) \text{ (trap (PR } privcmd)) \\
& (\text{CFG inputOK2 SM0StateInterp (certs2 cmd } npriv \text{ privcmd)} \\
& \quad (\text{Name Carol says prop (SOME (PR } privcmd)) :: ins) \ s \ outs) \\
& (\text{CFG inputOK2 SM0StateInterp (certs2 cmd } npriv \text{ privcmd)} \\
& \quad \text{ins (NS } s \text{ (trap (PR } privcmd))} \\
& \quad \text{(Out } s \text{ (trap (PR } privcmd)) :: outs)}) \Rightarrow \\
& (M, Oi, Os) \text{ sat prop NONE}
\end{aligned}$$

(4) [\[Carol\\_justified\\_privcmd\\_trap\\_thm\]](#)

$$\begin{aligned}
& \vdash \text{inputOK2 (Name Carol says prop (SOME (PR } privcmd)))} \wedge \\
& \text{CFGInterpret } (M, Oi, Os) \\
& (\text{CFG inputOK2 SM0StateInterp (certs2 cmd } npriv \text{ privcmd)} \\
& \quad (\text{Name Carol says prop (SOME (PR } privcmd)) :: ins) \ s \\
& \quad \text{outs}) \Rightarrow
\end{aligned}$$

```

TR (M, Oi, Os) (trap (PR privcmd))
  (CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)
    (Name Carol says prop (SOME (PR privcmd)) :: ins) s outs)
  (CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)
    ins (NS s (trap (PR privcmd)))
    (Out s (trap (PR privcmd)) :: outs))

```

## 17.4 Source Files

### 17.4.1 Holmakefile

```
INCLUDES=${HOME}/Documents/RESEARCH/HOL/ACL/
```

### 17.4.2 satListScript.sml

```

(* ----- *)
(* Definition of satList for conjunctions of ACL formulas *)
(* Author: Shiu-Kai Chin *)
(* Date: 24 July 2014 *)
(* ----- *)
structure satListScript = struct

(* interactive mode
  app load
  ["TypeBase", "listTheory", "acl_infRules"];
*)
open HolKernel boolLib Parse bossLib
open TypeBase acl_infRules listTheory

(* *****
  * create a new theory
  * ***** *)
val _ = new_theory "satList";

(* *****
  * Configurations and policies are represented by lists
  * of formulas in the access-control logic.
  * Previously, for a formula f in the access-control logic,
  * we ultimately interpreted it within the context of a
  * Kripke structure M and partial orders Oi: 'Int po and
  * Os: 'Sec po. This is represented as (M, Oi, Os) sat f.
  * The natural extension is to interpret a list of formulas
  * [f0; ...; fn] as a conjunction:
  * (M, Oi, Os) sat f0 /\ ... /\ (M, Oi, Os) sat fn
  * ***** *)

val _ = set_fixity "satList" (Infixr 540);

val satList_def =
Define
  '(M:('prop, 'world, 'pName, 'Int, 'Sec) Kripke), (Oi: 'Int po), (Os: 'Sec po))
  satList
  formList =
FOLDR
  (\x y. x /\ y) T
  (MAP
    (\ (f:('prop, 'pName, 'Int, 'Sec) Form).
      ((M:('prop, 'world, 'pName, 'Int, 'Sec) Kripke),
        Oi: 'Int po, Os: 'Sec po) sat f) formList)';

```

```

(*****)
(* Properties of satList *)
(*****)
val satList_nil =
TAC.PROOF(
([],
  '!(M:( 'prop , 'world , 'pName , 'Int , 'Sec) Kripke ),(Oi: 'Int po),(Os: 'Sec po)) satList [] ' '),
REWRITE_TAC[satList_def ,FOLDR,MAP])

val _ = save_thm("satList_nil",satList_nil)

val satList_conj =
TAC.PROOF(
([],
  '!!l1 l2 M Oi Os.(((M:( 'prop , 'world , 'pName , 'Int , 'Sec) Kripke ),(Oi: 'Int po),(Os: 'Sec po))
    satList l1) /\
    (((M:( 'prop , 'world , 'pName , 'Int , 'Sec) Kripke ),(Oi: 'Int po),(Os: 'Sec po))
      satList l2) =
    (((M:( 'prop , 'world , 'pName , 'Int , 'Sec) Kripke ),(Oi: 'Int po),(Os: 'Sec po))
      satList (l1 ++ l2)) ' '),
Induct THEN
REWRITE_TAC[APPEND,satList_nil] THEN
REWRITE_TAC[satList_def ,MAP] THEN
CONV_TAC(DEPTH.CONV BETA.CONV) THEN
REWRITE_TAC[FOLDR] THEN
CONV_TAC(DEPTH.CONV BETA.CONV) THEN
REWRITE_TAC[GSYM satList_def] THEN
PROVE_TAC[])

val _ = save_thm("satList_conj",satList_conj)

val satList_CONS =
TAC.PROOF(([],
  '!h t M Oi Os.(((M:( 'prop , 'world , 'pName , 'Int , 'Sec) Kripke ),(Oi: 'Int po),(Os: 'Sec po))
    satList (h::t)) =
    (((M,Oi,Os) sat h) /\
    (((M:( 'prop , 'world , 'pName , 'Int , 'Sec) Kripke ),(Oi: 'Int po),(Os: 'Sec po))
      satList t)) ' '),
REPEAT STRIP_TAC THEN
REWRITE_TAC[satList_def ,MAP] THEN
CONV_TAC(DEPTH.CONV BETA.CONV) THEN
REWRITE_TAC[FOLDR] THEN
CONV_TAC(DEPTH.CONV BETA.CONV) THEN
REWRITE_TAC[])

val _ = save_thm("satList_CONS",satList_CONS)

val _ = export_theory ();
val _ = print_theory "-";

end (* structure *)

```

### 17.4.3 ssminfRules.sml

```

(*****)
(* Inference rules for secure state-machines *)
(*****)
structure ssminfRules :> ssminfRules = struct

open HolKernel boolLib Parse bossLib
open reduceLib

fun flip_imp term =
let

```

```

    val (a,c) = dest_imp term
  in
    mk_imp(c,a)
  end

fun flip_TR_rules TR_rules =
let
  val thmlist = map SPEC_ALL (CONJUNCTS TR_rules)
  val conclist = map concl thmlist
  val implist = map flip_imp conclist
in
  list_mk_conj implist
end

fun TR_EQ_rules TR_rules TR_rules_converse =
let
  val th1list = (map SPEC_ALL (CONJUNCTS TR_rules))
  val th2list = CONJUNCTS TR_rules_converse
  val th3list = map2 IMP_ANTISYM_RULE th2list th1list
in
  LIST_CONJ th3list
end

end; (* structure *)

```

#### 17.4.4 ssminfRules.sig

```

signature ssminfRules = sig

type tactic = Abbrev.tactic;

type thm_tactic = Abbrev.thm_tactic;

type conv = Abbrev.conv;

type thm = Thm.thm;

type term = Term.term;

type hol_type = Type.hol_type

val flip_imp : term -> term

val flip_TR_rules : thm -> term

val TR_EQ_rules : thm -> thm -> thm

end;

```

#### 17.4.5 ssm1Script.sml

```

(*****
(* Secure State Machine Theory: authentication, authorization, and state *)
(* interpretation. *)
(* Author: Shiu-Kai Chin *)
(* Date: 27 November 2015 *)
*****)

structure ssm1Script = struct

(* ==== Interactive mode ====
app load ["TypeBase", "ssminfRules", "listTheory", "optionTheory", "acl_infRules",
         "satListTheory", "ssm1Theory"];
open TypeBase listTheory ssminfRules optionTheory acl_infRules satListTheory
      ssm1Theory

```

```

==== end interactive mode ==== *)

open HolKernel boolLib Parse bossLib
open TypeBase listTheory optionTheory ssminfRules acl_infRules satListTheory
(*****)
(* create a new theory *)
(*****)
val _ = new_theory "ssml";

(* ----- *)
(* Define the type of transition: discard, execute, or trap. We discard from *)
(* the input stream those inputs that are not of the form P says command. We *)
(* execute commands that users and supervisors are authorized for. We trap *)
(* commands that users are not authorized to execute. *)
(* ----- *)

(* ----- *)
(* In keeping with virtual machine design principles as described by Popek *)
(* and Goldberg, we add a TRAP instruction to the commands by users. *)
(* In effect, we are LIFTING the commands available to users to include the *)
(* TRAP instruction used by the state machine to handle authorization errors. *)
(* ----- *)

val _ =
  Datatype
  `inst = SOME `command | NONE`

val inst_distinct_clauses = distinct_of `:` `command inst`
val _ = save_thm("inst_distinct_clauses", inst_distinct_clauses)

val inst_one_one = one_one_of `:` `command inst`
val _ = save_thm("inst_one_one", inst_one_one)

val _ =
  Datatype
  `trType =
    discard | trap `command | exec `command`

val trType_distinct_clauses = distinct_of `:` `command trType`
val _ = save_thm("trType_distinct_clauses", trType_distinct_clauses)

val trType_one_one = one_one_of `:` `command trType`
val _ = save_thm("trType_one_one", trType_one_one)

(* ----- *)
(* Define configuration to include the security context within which the *)
(* inputs are evaluated. The components are as follows: (1) the authentication *)
(* function, (2) the interpretation of the state, (3) the security context, *)
(* (4) the input stream, (5) the state, and (6) the output stream. *)
(* ----- *)

val _ =
  Datatype
  `configuration =
    CFG
    ((`command inst, `principal, `d, `e)Form -> bool)
    ((`state -> (`command inst, `principal, `d, `e)Form))
    ((`command inst, `principal, `d, `e)Form list)
    ((`command inst, `principal, `d, `e)Form list)
    (`state)
    (`output list)`

(* ----- *)
(* Prove one-to-one properties of configuration *)
(* ----- *)

val configuration_one_one =

```

```

one_one_of '':('command inst,'d','e','output','principal','state)configuration ''

val _ = save_thm("configuration_one_one",configuration_one_one)

(* ----- *)
(* The interpretation of configuration is the conjunction of the formulas in *)
(* the context and the first element of a non-empty input stream. *)
(* ----- *)
val CFGInterpret_def =
Define
'CFGInterpret
(M:(('command inst','b','principal','d','e')Kripke),Oi:'d po,Os:'e po)
(CFG
(inputTest:('command inst','principal','d','e')Form -> bool)
(stateInterp:'state -> ('command inst','principal','d','e')Form)
(context:('command inst','principal','d','e')Form list)
((x:('command inst','principal','d','e')Form)::ins)
(state:'state)
(outStream:'output list))
=
((M,Oi,Os) satList context) /\
((M,Oi,Os) sat x) /\
((M,Oi,Os) sat (stateInterp state))

(* ----- *)
(* Define transition relation among configurations. This definition is *)
(* parameterized in terms of next-state transition function and output *)
(* function. *)
(* The first rule is set up with the expectation it is some principal P *)
(* ordering a command cmd be executed. *)
(* ----- *)
val (TR_rules, TR_ind, TR_cases) =
Hol_reln
'(! (inputTest:('command inst','principal','d','e')Form -> bool) (P:'principal Princ)
(NS:'state -> 'command trType -> 'state) M Oi Os Out (s:'state)
(certList:('command inst','principal','d','e')Form list)
(stateInterp:'state -> ('command inst','principal','d','e')Form)
(cmd:'command)(ins:('command inst','principal','d','e')Form list)
(outs:'output list).
(inputTest ((P says (prop (SOME cmd))):('command inst','principal','d','e')Form) /\
(CFGInterpret (M,Oi,Os)
(CFG inputTest stateInterp certList
(((P says (prop (SOME cmd))):('command inst','principal','d','e')Form)::ins)
s outs))) ==>
(TR
(M:(('command inst','b','principal','d','e')Kripke),Oi:'d po,Os:'e po) (exec cmd)
(CFG inputTest stateInterp certList
(((P says (prop (SOME cmd))):('command inst','principal','d','e')Form)::ins)
s outs)
(CFG inputTest stateInterp certList ins (NS s (exec cmd))
((Out s (exec cmd))::outs))) /\
(! (inputTest:('command inst','principal','d','e')Form -> bool) (P:'principal Princ)
(NS:'state -> 'command trType -> 'state) M Oi Os Out (s:'state)
(certList:('command inst','principal','d','e')Form list)
(stateInterp:'state -> ('command inst','principal','d','e')Form)
(cmd:'command)(ins:('command inst','principal','d','e')Form list)
(outs:'output list).
(inputTest ((P says (prop (SOME cmd))):('command inst','principal','d','e')Form) /\
(CFGInterpret (M,Oi,Os)
(CFG inputTest stateInterp certList
(((P says (prop (SOME cmd))):('command inst','principal','d','e')Form)::ins)
s outs))) ==>
(TR
(M:(('command inst','b','principal','d','e')Kripke),Oi:'d po,Os:'e po) (trap cmd)
(CFG inputTest stateInterp certList
(((P says (prop (SOME cmd))):('command inst','principal','d','e')Form)::ins)
s outs)

```



```

(CFG inputTest stateInterp certList ins (NS s (trap cmd))
  ((Out s (trap cmd)):: outs))) /\
(! (inputTest:('command inst,'principal,'d,'e)Form -> bool)
  (NS:'state -> 'command trType -> 'state)
  M Oi Os (Out: 'state -> 'command trType -> 'output) (s:'state)
  (certList:('command inst,'principal,'d,'e)Form list)
  (stateInterp:'state -> ('command inst,'principal,'d,'e)Form)
  (cmd:'command)(x:('command inst,'principal,'d,'e)Form)
  (ins:('command inst,'principal,'d,'e)Form list)
  (outs:'output list)).
~inputTest x ==>
(TR
  ((M:('command inst,'b,'principal,'d,'e)Kripke),Oi:'d po,Os:'e po)
  (discard:'command trType)
  (CFG inputTest stateInterp certList
    ((x:('command inst,'principal,'d,'e)Form):: ins) s outs)
  (CFG inputTest stateInterp certList ins (NS s discard)
    ((Out s discard):: outs)))) '

(* ----- *)
(* Split up TR_rules into individual clauses *)
(* ----- *)
val [rule0,rule1,rule2] = CONJUNCTS TR_rules

(*****
(* Prove the converse of rule0, rule1, and rule2 *)
*****)
val TR_lemma0 =
TAC.PROOF([], flip_TR_rules rule0),
DISCH.TAC THEN
IMP.RES.TAC TR_cases THEN
PAT.ASSUM
  ``exec cmd = y``
  (fn th => ASSUME.TAC(REWRITE.RULE[trType_one_one, trType_distinct_clauses]th)) THEN
PROVE.TAC[configuration_one_one, list_l1, trType_distinct_clauses]

val TR_lemma1 =
TAC.PROOF([], flip_TR_rules rule1),
DISCH.TAC THEN
IMP.RES.TAC TR_cases THEN
PAT.ASSUM
  ``trap cmd = y``
  (fn th => ASSUME.TAC(REWRITE.RULE[trType_one_one, trType_distinct_clauses]th)) THEN
PROVE.TAC[configuration_one_one, list_l1, trType_distinct_clauses]

val TR_lemma2 =
TAC.PROOF([], flip_TR_rules rule2),
DISCH.TAC THEN
IMP.RES.TAC TR_cases THEN
PAT.ASSUM
  ``discard = y``
  (fn th => ASSUME.TAC(REWRITE.RULE[trType_one_one, trType_distinct_clauses]th)) THEN
PROVE.TAC[configuration_one_one, list_l1, trType_distinct_clauses]

val TR_rules_converse =
TAC.PROOF([], flip_TR_rules TR_rules),
REWRITE.TAC[TR_lemma0, TR_lemma1, TR_lemma2])

val TR_EQ_rules_thm = TR_EQ_rules TR_rules TR_rules_converse

val _ = save_thm("TR_EQ_rules_thm", TR_EQ_rules_thm)

val [TRrule0, TRrule1, TR_discard_cmd_rule] = CONJUNCTS TR_EQ_rules_thm

val _ = save_thm("TRrule0", TRrule0)

```

```

val _ = save_thm("TRrule1",TRrule1)
val _ = save_thm("TR_discard_cmd_rule",TR_discard_cmd_rule)

(* ----- *)
(* If (CFGInterpret *)
(* (M,Oi,Os) *)
(* (CFG inputTest stateInterpret certList *)
(* ((P says (prop (CMD cmd))::ins) s outs) ==> *)
(* ((M,Oi,Os) sat (prop (CMD cmd)))) *)
(* is a valid inference rule, then executing cmd the exec(CMD cmd) transition *)
(* occurs if and only if prop (CMD cmd), inputTest, and *)
(* CFGInterpret (M,Oi,Os) *)
(* (CFG inputTest stateInterpret certList (P says prop (CMD cmd)::ins) s outs) *)
(* are true. *)
(* ----- *)
val TR_exec_cmd_rule =
TAC_PROOF([[] ,
"!inputTest certList stateInterp P cmd ins s outs.
(!M Oi Os.
(CFGInterpret
(M :('command inst, 'b, 'principal, 'd, 'e) Kripke),(Oi : 'd po), (Os : 'e po))
(CFG inputTest
(stateInterp : 'state -> ('command inst, 'principal, 'd, 'e)Form) certList
(P says (prop (SOME cmd) :('command inst, 'principal, 'd, 'e) Form)::ins)
(s : 'state) (outs : 'output list)) ==>
(M,Oi,Os) sat (prop (SOME cmd) :('command inst, 'principal, 'd, 'e) Form))) ==>
(!NS Out M Oi Os.
TR
(M :('command inst, 'b, 'principal, 'd, 'e) Kripke),(Oi : 'd po),
(Os : 'e po)) (exec (cmd : 'command))
(CFG (inputTest :('command inst, 'principal, 'd, 'e) Form -> bool)
(stateInterp : 'state -> ('command inst, 'principal, 'd, 'e)Form)
(certList :('command inst, 'principal, 'd, 'e) Form list)
((P : 'principal Princ) says
(prop (SOME cmd) :('command inst, 'principal, 'd, 'e) Form)::
(ins :('command inst, 'principal, 'd, 'e) Form list))
(s : 'state) (outs : 'output list))
(CFG inputTest stateInterp certList ins
((NS : 'state -> 'command trType -> 'state) s (exec cmd))
((Out : 'state -> 'command trType -> 'output) s (exec cmd)::
outs)) <=>
inputTest
(P says (prop (SOME cmd) :('command inst, 'principal, 'd, 'e) Form)) /\
(CFGInterpret (M,Oi,Os)
(CFG
inputTest stateInterp certList
(P says (prop (SOME cmd):('command inst, 'principal, 'd, 'e) Form)::ins)
s outs)) /\
(M,Oi,Os) sat (prop (SOME cmd) :('command inst, 'principal, 'd, 'e) Form))''),
REWRITE_TAC[TRrule0] THEN
REPEAT STRIP_TAC THEN
EQ_TAC THEN
REPEAT STRIP_TAC THEN
PROVE_TAC[])

val _ = save_thm("TR_exec_cmd_rule",TR_exec_cmd_rule)

(* ----- *)
(* If (CFGInterpret *)
(* (M,Oi,Os) *)
(* (CFG inputTest stateInterpret certList *)
(* ((P says (prop (CMD cmd))::ins) s outs) ==> *)
(* ((M,Oi,Os) sat (prop TRAP))) *)
(* is a valid inference rule, then executing cmd the exec(CMD cmd) transition *)
(* occurs if and only if prop TRAP, inputTest, and *)
(* CFGInterpret (M,Oi,Os) *)

```

```

(* (CFG inputTest stateInterpret certList (P says prop (CMD cmd)::ins) *)
(* s outs) are true. *)
*)
val TR_trap_cmd_rule =
TAC.PROOF(
([],
  '!'inputTest (stateInterp:'state -> ('command inst,'principal,'d,'e)Form) certList
    P cmd ins (s:'state) (outs:'output list).
    (!M Oi Os.
    CFGInterpret
      ((M :('command inst, 'b, 'principal, 'd, 'e) Kripke),(Oi :'d po), (Os :'e po))
      (CFG inputTest stateInterp certList
        (P says (prop (SOME cmd) :('command inst, 'principal, 'd, 'e) Form)::ins)
          s outs) ==>
        (M,Oi,Os) sat (prop NONE :('command inst, 'principal, 'd, 'e) Form)) ==>
      (!NS Out M Oi Os.
      TR
        ((M :('command inst, 'b, 'principal, 'd, 'e) Kripke),(Oi :'d po),
          (Os :'e po)) (trap (cmd :'command))
        (CFG (inputTest :('command inst, 'principal, 'd, 'e) Form -> bool)
          (stateInterp:'state -> ('command inst,'principal,'d,'e)Form)
          (certList :('command inst, 'principal, 'd, 'e) Form list)
          ((P :'principal Princ) says
            (prop (SOME cmd) :('command inst, 'principal, 'd, 'e) Form)::
              (ins :('command inst, 'principal, 'd, 'e) Form list))
          (s :'state) outs)
        (CFG inputTest (stateInterp:'state -> ('command inst,'principal,'d,'e)Form) certList ins
          ((NS :'state -> 'command trType -> 'state) s (trap cmd))
          ((Out :'state -> 'command trType -> 'output) s
            (trap cmd)::outs)) <=>
        inputTest
          (P says
            (prop (SOME cmd) :('command inst, 'principal, 'd, 'e) Form)) /\
          CFGInterpret (M,Oi,Os)
            (CFG inputTest (stateInterp:'state -> ('command inst,'principal,'d,'e)Form) certList
              (P says
                (prop (SOME cmd) :('command inst, 'principal, 'd, 'e) Form)::ins)
                s outs) /\
            (M,Oi,Os) sat (prop NONE)) ' '),
    REWRITE_TAC[TRrule1] THEN
    REPEAT STRIP_TAC THEN
    EQ_TAC THEN
    REPEAT STRIP_TAC THEN
    PROVE_TAC[])

val _ = save_thm("TR_trap_cmd_rule",TR_trap_cmd_rule)
(* ==== start here ====

==== end here ==== *)

val _ = export_theory ();
val _ = print_theory "-";

end (* structure *)

```

## 17.4.6 SM0Script.sml

```

(* ***** *)
(* Machine SM0 example *)
(* Author: Shiu-Kai Chin *)
(* Date: 30 November 2015 *)
(* ***** *)

structure SM0Script = struct

(* interactive mode

```

---

```

app load ["TypeBase","ssmlTheory","SM0Theory","acl_infRules","aclrulesTheory",
        "aclDrulesTheory","SM0Theory"];
open TypeBase ssmlTheory acl_infRules aclrulesTheory
    aclDrulesTheory satListTheory SM0Theory
*)

open HolKernel boolLib Parse bossLib
open TypeBase ssmlTheory acl_infRules aclrulesTheory aclDrulesTheory
    satListTheory

(*****
 * create a new theory
 *****)

val _ = new_theory "SM0"

(----- *)
(* Define datatypes for commands and their properties *)
(----- *)
val _ =
  Datatype 'privcmd = launch | reset '

val privcmd_distinct_clauses = distinct_of '':privcmd ''
val _ = save_thm("privcmd_distinct_clauses",privcmd_distinct_clauses)

val _ =
  Datatype 'npriv = status '

val _ =
  Datatype 'command = NP npriv | PR privcmd '

val command_distinct_clauses = distinct_of '':command ''
val _ = save_thm("command_distinct_clauses",command_distinct_clauses)

val command_one_one = one_one_of '':command ''
val _ = save_thm("command_one_one",command_one_one)

(----- *)
(* Define the states *)
(----- *)
val _ =
  Datatype 'state = STBY | ACTIVE '

val state_distinct_clauses = distinct_of '':state ''
val _ = save_thm("state_distinct_clauses",state_distinct_clauses)

(----- *)
(* Define the outputs *)
(----- *)
val _ =
  Datatype 'output = on | off '

val output_distinct_clauses = distinct_of '':output ''
val _ = save_thm("output_distinct_clauses",output_distinct_clauses)

(----- *)
(* Define next-state function for machine M0 *)
(----- *)
val SM0ns_def =
  Define
    '(SM0ns STBY (exec (PR reset)) = STBY) /\
     (SM0ns STBY (exec (PR launch)) = ACTIVE) /\
     (SM0ns STBY (exec (NP status)) = STBY) /\
     (SM0ns ACTIVE (exec (PR reset)) = STBY) /\

```

```

(SM0ns ACTIVE (exec (PR launch)) = ACTIVE) /\
(SM0ns ACTIVE (exec (NP status)) = ACTIVE) /\
(SM0ns STBY (trap (PR reset)) = STBY) /\
(SM0ns STBY (trap (PR launch)) = STBY) /\
(SM0ns STBY (trap (NP status)) = STBY) /\
(SM0ns ACTIVE (trap (PR reset)) = ACTIVE) /\
(SM0ns ACTIVE (trap (PR launch)) = ACTIVE) /\
(SM0ns ACTIVE (trap (NP status)) = ACTIVE) /\
(SM0ns STBY discard = STBY) /\
(SM0ns ACTIVE discard = ACTIVE)‘

(* ----- *)
(* Define next-output function for machine M0 *)
(* ----- *)
val SM0out_def =
Define
‘(SM0out STBY (exec (PR reset)) = off) /\
(SM0out STBY (exec (PR launch)) = on) /\
(SM0out STBY (exec (NP status)) = off) /\
(SM0out ACTIVE (exec (PR reset)) = off) /\
(SM0out ACTIVE (exec (PR launch)) = on) /\
(SM0out ACTIVE (exec (NP status)) = on) /\
(SM0out STBY (trap (PR reset)) = off) /\
(SM0out STBY (trap (PR launch)) = off) /\
(SM0out STBY (trap (NP status)) = off) /\
(SM0out ACTIVE (trap (PR reset)) = on) /\
(SM0out ACTIVE (trap (PR launch)) = on) /\
(SM0out ACTIVE (trap (NP status)) = on) /\
(SM0out STBY discard = off) /\
(SM0out ACTIVE discard = on)‘

(* ----- *)
(* Define datatypes for principles and their properties *)
(* ----- *)
val _ =
Datatype ‘staff = Alice | Bob | Carol‘

val staff_distinct_clauses = distinct_of ‘:staff‘
val _ = save_thm("staff_distinct_clauses", staff_distinct_clauses)

(* ----- *)
(* Input Authentication *)
(* ----- *)
val inputOK_def =
Define
‘(inputOK
  (((Name Alice) says
    (prop (SOME (cmd:command)))):(command inst, staff, 'd, 'e)Form) = T) /\
(inputOK
  (((Name Bob) says
    (prop (SOME (cmd:command)))):(command inst, staff, 'd, 'e)Form) = T) /\
(inputOK _ = F)‘

(* ----- *)
(* SM0StateInterp *)
(* ----- *)
val SM0StateInterp_def =
Define
‘SM0StateInterp (state:state) = (TT:(command inst, staff, 'd, 'e)Form)‘

(* ----- *)
(* certs definition *)
(* ----- *)
val certs_def =
Define
‘certs (cmd:command)(npriv:npriv)(privcmd:privcmd) =
  [(Name Alice controls ((prop (SOME (NP npriv)))):(command inst, staff, 'd, 'e)Form));

```

```

Name Alice controls (prop (SOME (PR privcmd)));
Name Bob controls prop (SOME (NP npriv));
((Name Bob) says (prop (SOME (PR privcmd)))) impf (prop NONE)]'

(* ----- *)
(* Some theorems showing any message from Carol is rejected *)
(* ----- *)
val Carol_rejected_lemma =
TAC.PROOF([],
  "'inputOK
    (((Name Carol) says (prop (SOME (cmd:command)))):(command inst ,staff , 'd, 'e)Form) ' ',
  PROVE_TAC[inputOK_def])

val _ = save_thm("Carol_rejected_lemma", Carol_rejected_lemma)

val Carol_discard_lemma =
TAC.PROOF([],
  "'TR ((M:(command inst , 'b, staff , 'd, 'e)Kripke), Oi, Os) discard
    (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
      (((Name Carol) says (prop (SOME (cmd:command)))):: ins)
      s (outs:output list))
    (CFG inputOK SM0StateInterp (certs cmd npriv privcmd) ins
      (SM0ns s discard) ((SM0out s discard):: outs)) ' ',
  PROVE_TAC[Carol_rejected_lemma , TR_discard_cmd_rule])

val _ = save_thm("Carol_discard_lemma", Carol_discard_lemma)

(* ----- *)
(* Alice authorized on any privileged command *)
(* ----- *)
val Alice_privcmd_lemma =
TAC.PROOF([],
  "'CFGInterpret ((M:(command inst , 'b, staff , 'd, 'e)Kripke), Oi, Os)
    (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
      (((Name Alice) says (prop (SOME (PR (privcmd:privcmd)))):: ins)
      s (outs:output list)) ==>
      ((M, Oi, Os) sat (prop (SOME (PR privcmd)))) ' ',
  REWRITE_TAC[CFGInterpret_def, certs_def, SM0StateInterp_def, satList_CONS,
    satList_nil, sat_TT] THEN
  PROVE_TAC[Controls])

val _ = save_thm("Alice_privcmd_lemma", Alice_privcmd_lemma)

val Alice_exec_privcmd_justified_thm =
let
  val th1 =
  ISPECL
  [ "'inputOK:(command inst , staff , 'd, 'e)Form -> bool ' ',
    "'(certs cmd npriv privcmd):(command inst , staff , 'd, 'e)Form list ' ',
    "'SM0StateInterp:state ->(command inst , staff , 'd, 'e)Form ' ',
    "'Name Alice ' ', 'PR privcmd ' ', 'ins:(command inst , staff , 'd, 'e)Form list ' ',
    "'s:state ' ', 'outs:output list ' ' ]
  TR_exec_cmd_rule
in
TAC.PROOF([],
  "'!(NS :state -> command trType -> state)
    (Out :state -> command trType -> output)
    (M :(command inst , 'b, staff , 'd, 'e) Kripke) (Oi : 'd po)
    (Os : 'e po).
  TR (M, Oi, Os) (exec (PR (privcmd :privcmd)))
  (CFG (inputOK :(command inst , staff , 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst , staff , 'd, 'e) Form)
    (certs (cmd :command) (npriv :npriv) privcmd :
      (command inst , staff , 'd, 'e) Form list)
    (Name Alice says

```

```

      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::
        (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
      (outs :output list))
    (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs cmd npriv privcmd :
        (command inst, staff, 'd, 'e) Form list) ins
      (NS s (exec (PR privcmd)))
      (Out s (exec (PR privcmd))::outs)) <=>
inputOK
  (Name Alice says
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M,Oi,Os)
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::ins) s outs) /\
  (M,Oi,Os) sat
  (prop (SOME (PR privcmd) :command inst) :
    (command inst, staff, 'd, 'e) Form)‘),
PROVE_TAC[th1,Alice_privcmd_lemma])
end

val _ = save_thm("Alice_exec_privcmd_justified_thm",Alice_exec_privcmd_justified_thm)

(* ----- *)
(* If Alice's privileged command was executed, then the request was verified. *)
(* ----- *)
val Alice_privcmd_verified_thm =
TAC.PROOF([[],‘!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi :‘d po)
  (Os :‘e po).
  TR (M,Oi,Os) (exec (PR (privcmd :privcmd)))
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::
        (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
      (outs :output list))
    (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs cmd npriv privcmd :
        (command inst, staff, 'd, 'e) Form list) ins
      (NS s (exec (PR privcmd)))
      (Out s (exec (PR privcmd))::outs)) ==>
  (M,Oi,Os) sat
  (prop (SOME (PR privcmd) :command inst) :
    (command inst, staff, 'd, 'e) Form)‘),
PROVE_TAC[Alice_exec_privcmd_justified_thm])

val _ = save_thm("Alice_privcmd_verified_thm",Alice_privcmd_verified_thm)

val Alice_justified_privcmd_exec_thm =
TAC.PROOF([[],‘!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi :‘d po)
  (Os :‘e po) cmd npriv privcmd ins s outs.

```

```

inputOK
(Name Alice says
  (prop (SOME (PR privcmd) :command inst) :
    (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M,Oi,Os)
(CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
  (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
  (certs cmd npriv privcmd :
    (command inst, staff, 'd, 'e) Form list)
  (Name Alice says
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form)::ins) s outs) ==>
TR (M,Oi,Os) (exec (PR (privcmd :privcmd)))
(CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
  (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
  (certs (cmd :command) (npriv :npriv) privcmd :
    (command inst, staff, 'd, 'e) Form list)
  (Name Alice says
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form)::
      (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
    (outs :output list))
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list) ins
    (NS s (exec (PR privcmd)))
    (Out s (exec (PR privcmd))::outs)) ' '),
PROVE_TAC[ Alice_exec_privcmd_justified_thm, inputOK_def, Alice_privcmd_lemma ]

val _ = save_thm("Alice_justified_privcmd_exec_thm", Alice_justified_privcmd_exec_thm)

(* ===== start here =====

(* ----- *)
(* Exercise 17.3.1 *)
(* Alice's non-privileged commands are executed and justified *)
(* ----- *)
Alice_npriv_lemma

''CFGInterpret ((M:(command inst, 'b, staff, 'd, 'e) Kripke), Oi, Os)
  (CFG inputOK SM0StateInterp (certs cmd npriv privcmd)
    (((Name Alice) says (prop (SOME (NP (npriv:npriv)))))::ins)
    s (outs:output list)) ==>
  ((M,Oi,Os) sat (prop (SOME(NP npriv)))) ' '

Alice_exec_npriv_justified_thm

''!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po).
TR (M,Oi,Os) (exec (NP (npriv :npriv)))
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (NP npriv) :command inst) :
        (command inst, staff, 'd, 'e) Form)::
        (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
      (outs :output list))
    (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs cmd npriv privcmd :
        (command inst, staff, 'd, 'e) Form list) ins

```



```

      (NS s (exec (NP npriv)))
      (Out s (exec (NP npriv)):: outs)) <=>
inputOK
  (Name Alice says
    (prop (SOME (NP npriv) :command inst) :
      (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M,Oi,Os)
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (NP npriv) :command inst) :
        (command inst, staff, 'd, 'e) Form):: ins) s outs) /\
(M,Oi,Os) sat
(prop (SOME (NP npriv) :command inst) :
  (command inst, staff, 'd, 'e) Form))

```

*Alice\_npriv\_verified\_thm*

```

  ‘!(NS :state -> command trType -> state)
    (Out :state -> command trType -> output)
    (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
    (Os : 'e po).
TR (M,Oi,Os) (exec (NP (npriv :npriv)))
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (NP npriv) :command inst) :
        (command inst, staff, 'd, 'e) Form)::
        (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
        (outs :output list))
    (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs cmd npriv privcmd :
        (command inst, staff, 'd, 'e) Form list) ins
      (NS s (exec (NP npriv)))
      (Out s (exec (NP npriv)):: outs)) ==>
(M,Oi,Os) sat
(prop (SOME (NP npriv) :command inst) :
  (command inst, staff, 'd, 'e) Form))

```

*Alice\_justified\_npriv\_exec\_thm*

```

  ‘!(NS :state -> command trType -> state)
    (Out :state -> command trType -> output)
    (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
    (Os : 'e po) cmd npriv privcmd ins s outs.
inputOK
  (Name Alice says
    (prop (SOME (NP npriv) :command inst) :
      (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M,Oi,Os)
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Alice says
      (prop (SOME (NP npriv) :command inst) :
        (command inst, staff, 'd, 'e) Form):: ins) s outs) ==>
TR (M,Oi,Os) (exec (NP (npriv :npriv)))
  (CFG (inputOK :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)

```



```

(SM0StateInterp : state → (command inst, staff, 'd, 'e) Form)
(certs (cmd : command) (npriv : npriv) privcmd :
  (command inst, staff, 'd, 'e) Form list)
(Name Bob says
  (prop (SOME (PR privcmd) : command inst) :
    (command inst, staff, 'd, 'e) Form)::
    (ins : (command inst, staff, 'd, 'e) Form list)) (s : state)
  (outs : output list))
(CFG (inputOK : (command inst, staff, 'd, 'e) Form → bool)
  (SM0StateInterp : state → (command inst, staff, 'd, 'e) Form)
  (certs cmd npriv privcmd :
    (command inst, staff, 'd, 'e) Form list) ins
  (NS s (trap (PR privcmd)))
  (Out s (trap (PR privcmd)):: outs)) ==>
(M,Oi,Os) sat
(prop NONE:
  (command inst, staff, 'd, 'e) Form))‘‘

```

*Bob\_justified\_privcmd\_trap\_thm*

```

‘‘!(NS : state → command trType → state)
  (Out : state → command trType → output)
  (M : (command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po) cmd npriv privcmd ins s outs.
inputOK
(Name Bob says
  (prop (SOME (PR privcmd) : command inst) :
    (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M,Oi,Os)
(CFG (inputOK : (command inst, staff, 'd, 'e) Form → bool)
  (SM0StateInterp : state → (command inst, staff, 'd, 'e) Form)
  (certs cmd npriv privcmd :
    (command inst, staff, 'd, 'e) Form list)
  (Name Bob says
    (prop (SOME (PR privcmd) : command inst) :
      (command inst, staff, 'd, 'e) Form)::ins) s outs) ==>
  TR (M,Oi,Os) (trap (PR (privcmd : privcmd)))
  (CFG (inputOK : (command inst, staff, 'd, 'e) Form → bool)
    (SM0StateInterp : state → (command inst, staff, 'd, 'e) Form)
    (certs (cmd : command) (npriv : npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Bob says
      (prop (SOME (PR privcmd) : command inst) :
        (command inst, staff, 'd, 'e) Form)::
        (ins : (command inst, staff, 'd, 'e) Form list)) (s : state)
      (outs : output list))
    (CFG (inputOK : (command inst, staff, 'd, 'e) Form → bool)
      (SM0StateInterp : state → (command inst, staff, 'd, 'e) Form)
      (certs cmd npriv privcmd :
        (command inst, staff, 'd, 'e) Form list) ins
      (NS s (trap (PR privcmd)))
      (Out s (trap (PR privcmd)):: outs))‘‘

```

```

(* ----- *)
(* Exercise 17.3.3A *)
(* inputOK2 and certs2 defined to authenticate Carol only. Carol is *)
(* authorized solely on npriv commands, and trapped on privcmd. *)
(* ----- *)
val inputOK2_def =

```

*val certs2\_def =*

```

(* ----- *)
(* Exercise 17.3.3 B *)
(* Carol can execute non-privileged commands using inputOK2 and certs2 *)
(* ----- *)

```

(*\** \_\_\_\_\_ *\**)  
*Carol\_npriv\_lemma*

```

“CFGInterpret ((M:(command inst, 'b, staff, 'd, 'e) Kripke), Oi, Os)
  (CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)
    (((Name Carol) says (prop (SOME (NP (npriv:npriv))))): ins)
    s (outs:output list)) ==>
  ((M, Oi, Os) sat (prop (SOME(NP npriv)))) “

```

*Carol\_exec\_npriv\_justified\_thm*

```

“!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po).
TR (M, Oi, Os) (exec (NP (npriv :npriv)))
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs2 (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Carol says
      (prop (SOME (NP npriv) :command inst) :
        (command inst, staff, 'd, 'e) Form))::
      (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
    (outs :output list))
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs2 cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list) ins
    (NS s (exec (NP npriv)))
    (Out s (exec (NP npriv))::outs)) <=>
inputOK2
  (Name Carol says
    (prop (SOME (NP npriv) :command inst) :
      (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M, Oi, Os)
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs2 cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Carol says
      (prop (SOME (NP npriv) :command inst) :
        (command inst, staff, 'd, 'e) Form))::ins) s outs) /\
(M, Oi, Os) sat
  (prop (SOME (NP npriv) :command inst) :
    (command inst, staff, 'd, 'e) Form)) “

```

*Carol\_npriv\_verified\_thm*

```

“!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po).
TR (M, Oi, Os) (exec (NP (npriv :npriv)))
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs2 (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Carol says
      (prop (SOME (NP npriv) :command inst) :
        (command inst, staff, 'd, 'e) Form))::
      (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
    (outs :output list))
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs2 cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list) ins

```

```

      (NS s (exec (NP npriv)))
      (Out s (exec (NP npriv)):: outs)) ==>
(M, Oi, Os) sat
(prop (SOME (NP npriv) :command inst) :
  (command inst, staff, 'd, 'e) Form))‘‘

```

*Carol\_justified\_npriv\_exec\_thm*

```

‘‘!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po) cmd npriv privcmd ins s outs.
inputOK2
(Name Carol says
  (prop (SOME (NP npriv) :command inst) :
    (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M, Oi, Os)
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs2 cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Carol says
      (prop (SOME (NP npriv) :command inst) :
        (command inst, staff, 'd, 'e) Form):: ins) s outs) ==>
TR (M, Oi, Os) (exec (NP (npriv :npriv)))
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs2 (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Carol says
      (prop (SOME (NP npriv) :command inst) :
        (command inst, staff, 'd, 'e) Form)::
        (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
      (outs :output list))
    (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
      (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
      (certs2 cmd npriv privcmd :
        (command inst, staff, 'd, 'e) Form list) ins
    (NS s (exec (NP npriv)))
    (Out s (exec (NP npriv)):: outs))‘‘

```

```

(* ----- *)
(* Exercise 17.3.3 C *)
(* Carol's request to execute a privileged command is trapped *)
(* ----- *)

```

*Carol\_privcmd\_trap\_lemma*

```

‘‘CFGInterpret ((M:(command inst, 'b, staff, 'd, 'e) Kripke), Oi, Os)
  (CFG inputOK2 SM0StateInterp (certs2 cmd npriv privcmd)
    (((Name Carol) says (prop (SOME (PR (privcmd:privcmd))))): ins)
    s (outs:output list)) ==>
(M, Oi, Os) sat (prop NONE))‘‘

```

*Carol\_trap\_privcmd\_justified\_thm*

```

‘‘!(NS :state -> command trType -> state)
  (Out :state -> command trType -> output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po).
TR (M, Oi, Os) (trap (PR (privcmd :privcmd)))
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
    (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
    (certs2 (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Carol says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::

```

```

      (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
      (outs :output list))
    (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form → bool)
      (SM0StateInterp :state → (command inst, staff, 'd, 'e) Form)
      (certs2 cmd npriv privcmd :
        (command inst, staff, 'd, 'e) Form list) ins
      (NS s (trap (PR privcmd)))
      (Out s (trap (PR privcmd))::outs)) <=>
inputOK2
  (Name Carol says
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M,Oi,Os)
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form → bool)
    (SM0StateInterp :state → (command inst, staff, 'd, 'e) Form)
    (certs2 cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Carol says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::ins) s outs) /\
(M,Oi,Os) sat (prop NONE) : (command inst, staff, 'd, 'e) Form‘‘

```

*Carol\_privcmd\_trapped\_thm*

```

‘‘!(NS :state → command trType → state)
  (Out :state → command trType → output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po).
TR (M,Oi,Os) (trap (PR (privcmd :privcmd)))
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form → bool)
    (SM0StateInterp :state → (command inst, staff, 'd, 'e) Form)
    (certs2 (cmd :command) (npriv :npriv) privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Carol says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::
        (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
      (outs :output list))
    (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form → bool)
      (SM0StateInterp :state → (command inst, staff, 'd, 'e) Form)
      (certs2 cmd npriv privcmd :
        (command inst, staff, 'd, 'e) Form list) ins
      (NS s (trap (PR privcmd)))
      (Out s (trap (PR privcmd))::outs)) ==>
(M,Oi,Os) sat
(prop NONE:
  (command inst, staff, 'd, 'e) Form)‘‘

```

*Carol\_justified\_privcmd\_trap\_thm*

```

‘‘!(NS :state → command trType → state)
  (Out :state → command trType → output)
  (M :(command inst, 'b, staff, 'd, 'e) Kripke) (Oi : 'd po)
  (Os : 'e po) cmd npriv privcmd ins s outs.
inputOK2
  (Name Carol says
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form)) /\
CFGInterpret (M,Oi,Os)
  (CFG (inputOK2 :(command inst, staff, 'd, 'e) Form → bool)
    (SM0StateInterp :state → (command inst, staff, 'd, 'e) Form)
    (certs2 cmd npriv privcmd :
      (command inst, staff, 'd, 'e) Form list)
    (Name Carol says
      (prop (SOME (PR privcmd) :command inst) :
        (command inst, staff, 'd, 'e) Form)::ins) s outs) ==>
TR (M,Oi,Os) (trap (PR (privcmd :privcmd)))

```

```

(CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
  (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
  (certs2 (cmd :command) (npriv :npriv) privcmd :
    (command inst, staff, 'd, 'e) Form list)
  (Name Carol says
    (prop (SOME (PR privcmd) :command inst) :
      (command inst, staff, 'd, 'e) Form)::
      (ins :(command inst, staff, 'd, 'e) Form list)) (s :state)
    (outs :output list))
(CFG (inputOK2 :(command inst, staff, 'd, 'e) Form -> bool)
  (SM0StateInterp :state -> (command inst, staff, 'd, 'e) Form)
  (certs2 cmd npriv privcmd :
    (command inst, staff, 'd, 'e) Form list) ins
  (NS s (trap (PR privcmd)))
  (Out s (trap (PR privcmd))::outs)) ‘ ‘

```

==== end here ==== \*)

```

val _ = export_theory ()
val _ = print_theory "-"

```

**end** (\* structure \*)

BLANK PAGE



## Secure State Machine Refinements

---

BLANK PAGE

**Part VII**

**Appendix**



# Using Holmake

A critical capability is for people, other than the creators of a theory or theories, to rapidly reproduce theories, definitions, theorems, and proofs. This capability is no different that the use of VLSI (very large scale integrated) CAD (computer-aided design) tools to rapidly reproduced chip designs, layouts, schematics, simulations, and verifications.

In HOL, this is done using HOLMAKE. We will illustrate a simple example of its use first. Later on, we will show a more complicated example. Details of HOLMAKE are found in the HOL DESCRIPTION manual in the section named *Maintaining HOL Formalizations with Holmake*.

## 19.1 A Simple Example

In this simple example, we show how theorems we prove are stored as part of a named theory. For this simple case, we do not introduce any new datatypes or definitions.

The two theorems we prove are:

```
[theorem1]
  ⊢ ∀x y. x ∧ y ⇔ y ∧ x

[theorem2]
  ⊢ ∀x y z. x ∧ y ⇒ z ⇔ x ⇒ y ⇒ z
```

Their proofs are shown in the session below.

1

```
-----
HOL-4 [Kananaskis 9 (stdknl, built Sat Apr 26 13:42:15 2014)]

For introductory HOL help, type: help "hol";
To exit type <Control>-D (*not* quit());
-----

[loading theories and proof tools ..... ]
[closing file "/usr/local/share/HOL/tools/end-init-boss.sml"]

- ** Unicode trace now off
- val theorem1 = PROVE [] ``!x y.x /\ y = y /\ x``;
Meson search level: .....
> val theorem1 =
  |- !x y. x /\ y <=> y /\ x
  : thm
- val theorem2 =
TAC_PROOF(
  ([], ``!x y z.(x /\ y ==> z) = (x ==> y ==> z)``),
  PROVE_TAC []);
Meson search level: .....
> val theorem2 =
  |- !x y z. x /\ y ==> z <=> x ==> y ==> z
  : thm
```

Suppose we wish to do the following:

- include theorem1 and theorem2 in a theory called *exampleTheory*, and
- enable users to compile the source code to reproduce rapidly *exampleTheory*. This allows users to re-verify, re-build, and use the results in *exampleTheory*.

To do the above, we embed our proofs within a *script* file. Figure 19.1 is the script file that does what we want. Notice the following:

1. The name of the theory is part of the file name of the script file, i.e., `exampleScript.sml`.
2. The beginning of the script file starts with the `structure` statement at the top of `exampleScript.sml` and ends with `end` in the very last line of the script file.
3. The second part of the scriptfile opens some standard HOL theories, structures, and libraries, in this case *HolKernel*, *boolLib*, *Parse*, and *bossLib*. This is part of the `open` command.
4. The third part of the scriptfile is the `new_theory` function applied to ```example```.
5. The main body of the theory is next. In this case it is the two proofs of theorem1 and theorem2. **Note that after each proof the theorem is given a name and then stored in *exampleTheory* by the function `save_thm`.**
6. Finally, we print and export the theory as shown in Figure 19.1.

To build *exampleTheory* using `exampleScript.sml` we do the following in the script: (1) execute `Holmake cleanAll` to remove all old files and hidden subdirectories, followed by (2) `Holmake` to build the theory from scratch.

```
skchin@skVM:~/Documents/COURSES/ACE14/Appendix/HOL$ ls
CVS          exampleTheory.sig  exampleTheory.ui  HOLReports
exampleScript.sml  exampleTheory.sml  exampleTheory.uo
skchin@skVM:~/Documents/COURSES/ACE14/Appendix/HOL$ Holmake cleanAll
skchin@skVM:~/Documents/COURSES/ACE14/Appendix/HOL$ ls
CVS  exampleScript.sml  HOLReports
skchin@skVM:~/Documents/COURSES/ACE14/Appendix/HOL$ Holmake
Analysing exampleScript.sml
Trying to create directory .HOLMK for dependency files
Compiling exampleScript.sml
Linking exampleScript.uo to produce theory-builder executable
<<HOL message: Created theory "example">>
Saved theorem _____ "theorem1"
Saved theorem _____ "theorem2"
Theory: example

Parents:
  list

Theorems:
  theorem1
    |- x y. x y y x
  theorem2
    |- x y z. x y z x y z
Exporting theory "example" ... done.
Theory "example" took 0.497s to build
Analysing exampleTheory.sml
Analysing exampleTheory.sig
Compiling exampleTheory.sig
Compiling exampleTheory.sml
skchin@skVM:~/Documents/COURSES/ACE14/Appendix/HOL$
```

## 19.2 Using Theories and Libraries in Other Subdirectories

Often, developing new designs and theories depends on previous work. This means when we compile our HOL theories, we need to access theories in other subdirectories besides the subdirectory in which HOL is invoked.

**Figure 19.1** Example Theory Script

---

```

(* ===== *)
(* A simple example of creating a theory that is intended to be compiled *)
(* using Holmake *)
(* Author: Shiu-Kai Chin Date: 15 June 2014 *)
(* ===== *)

(* ----- *)
(* First, declare the file (exampleScript.sml) as a structure. We always *)
(* need to do this by naming the file, e.g., fooScript.sml *)
(* ----- *)
structure exampleScript = struct

(* ----- *)
(* Second, we open up some standard HOL theories, structures, and libraries. *)
(* This is also where you specify what existing theories you need to be *)
(* loaded and opened upon which your theory depends. In this case, we just *)
(* use the standard HolKernel, boolLib, Parse, and bossLib. *)
(* ----- *)
open HolKernel boolLib Parse bossLib

(* ----- *)
(* Name the new theory. Note, the name must be consistent with the script *)
(* file name. In this case, it is "example". *)
(* ----- *)
val _ = new_theory "example"

(* ----- *)
(* As an example, we prove a simple theorem using forward inference. *)
(* We name the theorem theorem1. *)
(* ----- *)
val theorem1 = PROVE [] ‘‘!x y.x /\ y = y /\ x‘‘

(* ----- *)
(* Next, we store theorem1 in exampleTheory under the name "theorem1" *)
(* ----- *)
val theorem1 =
  save_thm("theorem1",theorem1)

(* ----- *)
(* We prove another theorem, which we name theorem2, using TAC.PROOF and *)
(* PROVE.TAC. *)
(* ----- *)
val theorem2 =
  TAC.PROOF(
    ([], ‘‘!x y z.(x /\ y ==> z) = (x ==> y ==> z)‘‘),
    PROVE.TAC [])

(* ----- *)
(* Next, we store theorem2 in exampleTheory under the name "theorem2" *)
(* ----- *)
val theorem2 =
  save_thm("theorem2",theorem2)

(* ----- *)
(* Print and export the theory. *)
(* ----- *)
val _ = print_theory "-"

val _ = export_theory()

end (* structure *)

```

---



# Documentation for ML and HOL

---

Extensive documentation exists for ML and HOL. In this chapter we show you where to download the documentation for Moscow ML, upon which HOL is built. HOL is a massive system with extensive documentation. This documentation includes the following manuals and guides:

- **INTERACTION:** a short guide to using HOL for complete beginners
- **QUICK:** a quick reference sheet for commonly-used HOL inference rules, tactics, and conversions. This is handy to point you in the right direction as to what functions might be of use when attempting a proof.
- **TUTORIAL:** a more extensive introduction to building and using HOL. The tutorial includes case studies.
- **DESCRIPTION:** a detailed user's guide for HOL.
- **REFERENCE:** as the name implies, an encyclopedia of all the functions that are the HOL system.

For experienced users who have an idea of what they are searching for in HOL, there is a browser interface to HOL libraries, theories, structures, syntax, simplification sets, and theory bindings.

All of the above is available online at [hol.sourceforge.net](http://hol.sourceforge.net). In the case of HOL, the  $\text{\LaTeX}$  sources for the manuals and guides are included with the HOL sources. We will describe how to build them. The browser-accessible documentation is included with the HOL sources, too. This is convenient when no online access is available.

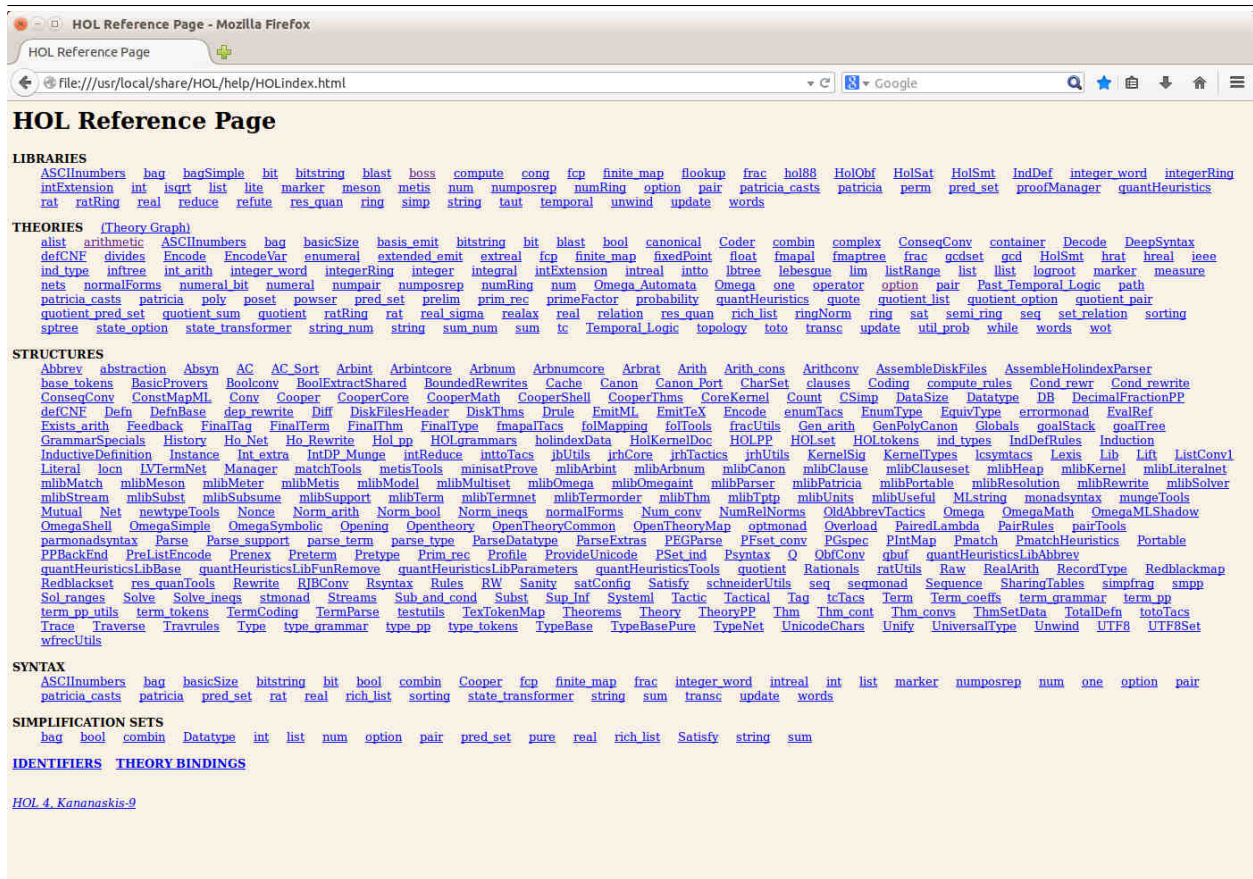
## 20.1 Documentation for Moscow ML

ML is a full-featured functional programming language with many libraries and examples. The documentation and source code for Moscow ML is at [mosml.org](http://mosml.org).

## 20.2 Documentation for HOL

In this section, we show you how to access the HOL documentation on-line and how to build the documentation locally.

**On-line documentation** All the on-line documentation (as well as the sources for building HOL) are available at [hol.sourceforge.net](http://hol.sourceforge.net). This includes all manuals and the browser-compatible interface to the HOL Reference Manual, as shown in Figure 20.1.

**Figure 20.1** HOL Reference Page

**Accessing the browser-compatible interface locally** The browser-compatible interface (as shown in Figure 20.1) to the HOL Reference Manual is accessible by double-clicking on the file:

```
/usr/local/share/HOL/help/HOLIndex.html.
```

If you know the name of the function you are looking for and want to learn what it does, the IDENTIFIERS link on the bottom left of the reference page. The HOL theories that are part of the distribution are listed under THEORIES. For experienced HOL users, this index is used often and is worth bookmarking in your browser.

**Building the HOL Manuals** The source distribution of HOL includes the  $\text{\LaTeX}$  sources for the major HOL manuals. The session below shows you how to build the manuals from sources. The `Makefile` for the sources is in the subdirectory

```
/usr/local/share/HOL/Manual
```

In your *terminal*, go to the subdirectory and execute:

```
sudo make clean
```

followed by

```
sudo make all ↵
```

The results are shown below. **Remember, make sure you do make as a superuser, since the subdirectories containing the manuals are owned by root.**

```
skchin@skVM:~$ cd /usr/local/share/HOL/Manual/
skchin@skVM:/usr/local/share/HOL/Manual$ ls
Description  Interaction  Logic  Makefile  README      Translations
Guide        LaTeX       Logo   Quick     Reference   Tutorial
skchin@skVM:/usr/local/share/HOL/Manual$ sudo make clean
[sudo] password for skchin:

..... lots of output .....

skchin@skVM:/usr/local/share/HOL/Manual$ sudo make all

..... lots of output .....

=====> MANUAL made
skchin@skVM:/usr/local/share/HOL/Manual$
```

After the manuals are built, their respective pdf files are available for viewing locally.

BLANK PAGE

## Summary of the Access-Control Logic

This appendix provides a summary of the syntax and inference rules of the access-control logic. It is intended as a canonical reference for the syntax and rules (both core and derived) introduced throughout the book.

### 21.1 Syntax

We define **PName** to be the collection of all simple principal names. The set **Princ** of all principal expressions is given by the following BNF specification:

$$\mathbf{Princ} ::= \mathbf{PName} / \mathbf{Princ} \ \& \ \mathbf{Princ} / \mathbf{Princ} \mid \mathbf{Princ}$$

The convention for compound principals is that  $\&$  binds more tightly than  $\mid$ .

We let **PropVar** be the collection of all propositional variables. The set **Form** of all well-formed expressions is given by the following BNF specification:

$$\begin{aligned} \mathbf{Form} ::= & \mathbf{PropVar} / \neg \mathbf{Form} / (\mathbf{Form} \vee \mathbf{Form}) / \\ & (\mathbf{Form} \wedge \mathbf{Form}) / (\mathbf{Form} \supset \mathbf{Form}) / (\mathbf{Form} \equiv \mathbf{Form}) / \\ & (\mathbf{Princ} \Rightarrow \mathbf{Princ}) / (\mathbf{Princ} \text{ says } \mathbf{Form}) / (\mathbf{Princ} \text{ controls } \mathbf{Form}) \\ & (\mathbf{Princ} \text{ reps } \mathbf{Princ} \text{ on } \varnothing) \end{aligned}$$

Parentheses can be omitted according to the following conventions for operator precedence, in decreasing tightness of bindings:

$$\begin{array}{c} \neg \\ \text{says} \quad \text{controls} \quad \text{reps} \\ \wedge \\ \vee \\ \supset \\ \equiv \end{array}$$

The definition of **Form** defines the core syntax of the logic. Three extensions are made to describe confidentiality, integrity, and role-based access control policies.

**Confidentiality** We define **SecLabel** to be the collection of *simple security labels*, which are used as names for the various levels associated with confidentiality. In addition to these specific security labels, we will often want to refer abstractly to the security level assigned to a particular principal  $P$ . For this reason, we define the larger set **SecLevel** of *all* possible security-level expressions:

$$\mathbf{SecLevel} ::= \mathbf{SecLabel} / \text{slev}(\mathbf{PName})$$

That is, a security-level expression is either a simple security label or an expression of the form  $\text{slev}(A)$ , where  $A$  is a simple principal name.<sup>1</sup> Informally,  $\text{slev}(A)$  refers to the security level of principal  $A$ .

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

$$\mathbf{Form} ::= \mathbf{SecLevel} \leq_s \mathbf{SecLevel} / \mathbf{SecLevel} =_s \mathbf{SecLevel}$$

**Integrity** We define **IntLabel** to be the collection of *simple integrity labels*, and we define **IntLevel** to be the set of *all* possible integrity-level expressions:

$$\mathbf{IntLevel} ::= \mathbf{IntLabel} / \text{ilev}(\mathbf{PName})$$

Informally,  $\text{ilev}(A)$  refers to the integrity—i.e., quality or trustworthiness—level of principal  $A$ .

We then extend our definition of well-formed formulas to support comparisons of security levels:

$$\mathbf{Form} ::= \mathbf{IntLevel} \leq_i \mathbf{IntLevel} / \mathbf{IntLevel} =_i \mathbf{IntLevel}$$

The symbol  $\leq_i$  denotes a partial ordering on integrity levels, in the same way that  $\leq_s$  denotes a partial ordering on security levels. In particular,  $\leq_i$  is reflexive, transitive, and antisymmetric.

**Role-based access control** We extend the syntax of the logic to accommodate statements that express equality among principals:

$$\mathbf{Form} ::= (\mathbf{Princ} = \mathbf{Princ})$$

## 21.2 Core Rules, Derived Rules, and Extensions

The following figures summarize the core rules, derived rules, and extensions for the access-control logic:

- Figure 21.1 is a summary of the core inference rules.
- Figure 21.2 is a summary of frequently used derived inference rules.
- Figure 21.3 is a summary of inference rules for delegation.
- Figure 21.4 is a summary of inference rules for relating security levels.
- Figure 21.5 is a summary of inference rules for relating integrity levels.
- Figure 21.6 is a summary of inference rules regarding principal equality.

<sup>1</sup>This syntax precludes security-level expressions such as  $\text{slev}(P \ \& \ Q)$  or  $\text{slev}(P \ | \ Q)$ , because there is no standard technique for associating security classification labels with compound principals.

---

**Figure 21.1** Summary of core rules for the access-control logic
 

---

$$\begin{array}{l}
 \textit{Taut} \quad \frac{}{\varphi} \quad \text{if } \varphi \text{ is an instance of a prop-logic tautology} \\
 \\
 \textit{Modus Ponens} \quad \frac{\varphi \quad \varphi \supset \varphi'}{\varphi'} \quad \textit{Says} \quad \frac{\varphi}{P \text{ says } \varphi} \\
 \\
 \textit{MP Says} \quad \frac{}{(P \text{ says } (\varphi \supset \varphi')) \supset (P \text{ says } \varphi \supset P \text{ says } \varphi')} \\
 \\
 \textit{Speaks For} \quad \frac{}{P \Rightarrow Q \supset (P \text{ says } \varphi \supset Q \text{ says } \varphi)} \\
 \\
 \textit{\& Says} \quad \frac{}{(P \& Q \text{ says } \varphi) \equiv ((P \text{ says } \varphi) \wedge (Q \text{ says } \varphi))} \\
 \\
 \textit{Quoting} \quad \frac{}{(P \mid Q \text{ says } \varphi) \equiv (P \text{ says } Q \text{ says } \varphi)} \\
 \\
 \textit{Idempotency of } \Rightarrow \quad \frac{}{P \Rightarrow P} \\
 \\
 \textit{Transitivity of } \Rightarrow \quad \frac{P \Rightarrow Q \quad Q \Rightarrow R}{P \Rightarrow R} \quad \textit{Monotonicity of } \Rightarrow \quad \frac{P \Rightarrow P' \quad Q \Rightarrow Q'}{P \mid Q \Rightarrow P' \mid Q'} \\
 \\
 \textit{Equivalence} \quad \frac{\varphi_1 \equiv \varphi_2 \quad \Psi[\varphi_1/q]}{\Psi[\varphi_2/q]} \\
 \\
 P \text{ controls } \varphi \quad \stackrel{\text{def}}{=} \quad (P \text{ says } \varphi) \supset \varphi
 \end{array}$$


---

**Figure 21.2** Summary of useful derived rules

<i>Conjunction</i>		$\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2}$
<i>Simplification (1)</i>	$\frac{\varphi_1 \wedge \varphi_2}{\varphi_1}$	<i>Simplification (2)</i> $\frac{\varphi_1 \wedge \varphi_2}{\varphi_2}$
<i>Disjunction (1)</i>	$\frac{\varphi_1}{\varphi_1 \vee \varphi_2}$	<i>Disjunction (2)</i> $\frac{\varphi_2}{\varphi_1 \vee \varphi_2}$
<i>Modus Tollens</i>	$\frac{\varphi_1 \supset \varphi_2 \quad \neg \varphi_2}{\neg \varphi_1}$	<i>Double negation</i> $\frac{\neg \neg \varphi}{\varphi}$
<i>Disjunctive Syllogism</i>	$\frac{\varphi_1 \vee \varphi_2 \quad \neg \varphi_1}{\varphi_2}$	<i>Hypothetical Syllogism</i> $\frac{\varphi_1 \supset \varphi_2 \quad \varphi_2 \supset \varphi_3}{\varphi_1 \supset \varphi_3}$
<i>Controls</i>		$\frac{P \text{ controls } \varphi \quad P \text{ says } \varphi}{\varphi}$
<i>Derived Speaks For</i>	$\frac{P \Rightarrow Q \quad P \text{ says } \varphi}{Q \text{ says } \varphi}$	<i>Derived Controls</i> $\frac{P \Rightarrow Q \quad Q \text{ controls } \varphi}{P \text{ controls } \varphi}$
<i>Says Simplification (1)</i>	$\frac{P \text{ says } (\varphi_1 \wedge \varphi_2)}{P \text{ says } \varphi_1}$	<i>Says Simplification (2)</i> $\frac{P \text{ says } (\varphi_1 \wedge \varphi_2)}{P \text{ says } \varphi_2}$

**Figure 21.3** Summary of rules for delegation

$P \text{ reps } Q \text{ on } \varphi \stackrel{\text{def}}{=} P \mid Q \text{ says } \varphi \supset Q \text{ says } \varphi$	
<i>Reps</i>	$\frac{Q \text{ controls } \varphi \quad P \text{ reps } Q \text{ on } \varphi \quad P \mid Q \text{ says } \varphi}{\varphi}$
<i>Rep Controls</i>	$\overline{A \text{ reps } B \text{ on } \varphi \equiv (A \text{ controls } (B \text{ says } \varphi))}$
<i>Rep Says</i>	$\frac{A \text{ reps } B \text{ on } \varphi \quad A \mid B \text{ says } \varphi}{B \text{ says } \varphi}$



**Figure 21.4** Inference rules for relating security levels

$$\ell_1 =_s \ell_2 \stackrel{\text{def}}{=} (\ell_1 \leq_s \ell_2) \wedge (\ell_2 \leq_s \ell_1)$$

$$\text{Reflexivity of } \leq_s \quad \overline{\ell \leq_s \ell}$$

$$\text{Transitivity of } \leq_s \quad \frac{\ell_1 \leq_s \ell_2 \quad \ell_2 \leq_s \ell_3}{\ell_1 \leq_s \ell_3}$$

**Figure 21.5** Inference rules for relating integrity levels

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

$$\text{Reflexivity of } \leq_i \quad \overline{\ell \leq_i \ell}$$

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

**Figure 21.6** Logical rules regarding principal equality

$$P = Q \stackrel{\text{def}}{=} P \Rightarrow Q \wedge Q \Rightarrow P$$

$$\text{Principal Equality} \quad \frac{P = Q \quad \varphi[P/A]}{\varphi[Q/A]}$$

$$\text{Distributivity of } | \quad \overline{P | (R_1 \& \dots \& R_k) = (P | R_1) \& \dots \& (P | R_k)} \quad (k \geq 1)$$

$$\text{Quoting Simplification} \quad \frac{P | (Q \& R) \text{ says } \varphi}{P | Q \text{ says } \varphi}$$

BLANK PAGE

---

# Bibliography

---

- [IEE, 1998] (1998). IEEE Guide for Information Technology–System Definition–Concept of Operations (ConOps) Document. IEEE Computer Society, IEEE Std 1362-1998.
- [JP5, 2011] (2011). JP 5-0, Joint Operation Planning. US Dept of Defense.
- [HOL, 2015a] (2015a). *The HOL System Description*. Available at [hol.sourceforge.net/documentation.html](http://hol.sourceforge.net/documentation.html).
- [HOL, 2015b] (2015b). *The HOL System Logic*. Available at [hol.sourceforge.net/documentation.html](http://hol.sourceforge.net/documentation.html).
- [HOL, 2015c] (2015c). *The HOL System Reference*. Available at [hol.sourceforge.net](http://hol.sourceforge.net) under Documentation.
- [Abadi et al., 1993] Abadi, M., Burrows, M., Lampson, B., and Plotkin, G. (1993). A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734.
- [Chin, 2015] Chin, S.-K. (2015). Teaching undergraduates certified security by design. In *19<sup>th</sup> Colloquium for Information Systems Security Education*, Las Vegas, NV.
- [Chin and Older, 2010] Chin, S.-K. and Older, S. (2010). *Access Control, Security, and Trust: A Logical Approach*. CRC Press, Boca Raton, FL.
- [Gordon and Melham, 1993] Gordon, M. and Melham, T. (1993). *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press.
- [Older and Chin, 2002] Older, S. and Chin, S.-K. (2002). Building a rigorous foundation for assurance into information assurance education. *George Washington University Journal of Information Security*, 1(2).
- [Older and Chin, 2012] Older, S. and Chin, S.-K. (2012). Engineering assurance at the undergraduate level. *IEEE Security and Privacy*, 10(6).
- [Paulson, 1996] Paulson, L. C. (1996). *ML for the Working Programmer*. Cambridge University Press.
- [Popek and Goldberg, 1974] Popek, G. J. and Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421.
- [Saltzer and Schroeder, 1975] Saltzer, J. and Schroeder, M. (1975). The Protection of Information in Computer Systems. *Proceedings IEEE*.